Broadview®
www.broadview.com.cn

Kubernetes 实战

吴龙辉 著

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

吴龙辉

网宿科技云计算架构师，负责云平台的设计和研发工作。活跃于CloudFoundry、Docker、Kubernetes等开源社区，致力于云计算PaaS的研究，拥有丰富的云计算实践经验。

Kubernetes 实战

吴龙辉 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Docker 的流行激活了一直不温不火的 PaaS，随之而来的是各类 Micro-PaaS 的出现，Kubernetes 是其中最具代表性的一员，它是 Google 多年大规模容器管理技术的开源版本。越来越多的企业被迫面对互联网规模所带来的各类难题，而 Kubernetes 以其优秀的理念和设计正在逐步形成新的技术标准，对于任何领域的运营总监、架构师和软件工程师来说，都是一个绝佳的突破机会。本书以理论加实战的模式，结合大量案例由浅入深地讲解了 Kubernetes 的各个方面，包括平台架构、基础核心功能、网络、安全和资源管理以及整个生态系统的组成，旨在帮助读者全面深入地掌握 Kubernetes+Docker 的底层技术堆栈。

图书在版编目（CIP）数据

Kubernetes 实战/吴龙辉著.—北京：电子工业出版社，2016.5
ISBN 978-7-121-28372-7

Ⅰ. ①K… Ⅱ. ①吴… Ⅲ. ①Linux 操作系统－程序设计 Ⅳ. ①TP316.85

中国版本图书馆 CIP 数据核字（2016）第 055126 号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787 × 980　1/16　　印张：17.75　　字数：355 千字
版　　次：2016 年 5 月第 1 版
印　　次：2016 年 5 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

前言

随着互联网技术在各领域的广泛应用，所产生的海量数据催生了大数据的诞生。而对于数据中心的需求激活了云计算井喷式的发展，一时间大数据和云计算成为各个企业争夺的战略高地。

在云计算领域的服务模式中，IaaS 和 SaaS 模式已经趋于成熟，因此 PaaS 就成了全球各大 IT 巨头和初创公司的焦点，其中的竞争异常激烈。大量的 PaaS 平台出现，又很快被淘汰，整个行业发生着巨大的迭代更替。正所谓物竞天择，在这样一个激荡变化的背景下，以 Docker 为代表的容器技术脱颖而出并极速发热，风头无两，大多数主流云厂商已经宣布提供对 Docker 及其生态系统的支持。容器技术具备融合 DevOps 的敏捷特性，给云计算市场特别是 PaaS 市场带来了新的变革力量，Kubernetes 就是新一轮变革中产生的一个代表性产品。

Kubernetes 是 Google 开源的容器集群管理系统，它对于容器运行时、编排、常规服务都抽象设计出了准确完整的 API，并以此建立起一个开放开源的系统，符合企业化需求，每家企业都可以以此搭建出自动化和标准化的底层平台，以优化研发和运营效率。Kubernetes 可以说是 Google 借助着容器领域的爆发，对于其巨大规模数据中心管理的丰富经验的一次实践，旨在建立新的技术业界标准。

展望未来，我们认为将有更多的企业被迫面对互联网规模所带来的各类难题，Kubernetes 和 Docker 技术可以提供应对这些挑战的解决方案。而随着更多企业的加入，会有更多的人以协作方式构建出更强大的技术堆栈和更多的创新成果，整个行业将朝着更好的方向持续迈进，对此我们乐观其成。

本书特点

本书采用的是理论加实战的模式，结合大量案例由浅入深讲解 Kubernetes 的各个方面，包括平台架构、基础核心功能、网络、安全和资源管理，以及整个生态系统的组成。技术信息完全来源于 Kubernetes 开源社区的文档、代码的提炼和总结。本书涉及的 Kubernetes 内容与官方最新版本同步，包含最新版本的所有新特性说明，并且因为 Kubernetes 同 Docker 深度集成，所以本书也会阐述 Docker 相关的技术话题。

本书的读者对象

本书适用于希望学习和使用 Kubernetes 以及正在寻找管理数据中心解决方案的软件工程师和架构师，同时本书可以作为 Docker 的高级延伸书籍，用于搭建基于 Kubernetes+Docker 的 PaaS 平台，实践 DevOps。

本书的组织结构

本书在组织结构上分成三部分：Kubernetes 基础篇、Kubernetes 高级篇和 Kubernetes 生态篇。基础篇可帮助读者认识 Kubernetes，并理解其架构和核心概念，同时能够部署和使用 Kubernetes 完成基本功能操作。高级篇将深入讲解 Kubernetes 的网络、安全和资源管理等话题，帮助读者掌握管理 Kubernetes 的能力。生态篇则介绍与 Kubernetes 密切相关的开源软件，包括 CoreOS、Etcd 和 Mesos，使读者对于 Kubernetes 生态系统有全面的了解。

目录

第 1 部分　Kubernetes 基础篇

第 3 部分 Kubernetes 生态篇

第 1 部分

Kubernetes 基础篇

第 1 章

Kubernetes 介绍

Kubernetes 可以说是云计算 PaaS 领域的集大成者，它借助了最好的帮助，并且在最适当的时间推出，从而得到了最多的关注。本章首先从整个行业背景介绍入手，介绍 Kubernetes 诞生的前因，并阐述 Kubernetes 的优势和发展历程，最后说明 Kubernetes 中的基本概念，帮助读者对 Kubernetes 有一个初步但全面的认识。

1.1 为什么会有 Kubernetes

1.1.1 云计算大潮

云计算（Cloud Computing）作为一个新兴领域，它是多种技术混合演进的结果，在许多大公司和初创企业的共同推动下，发展极为迅速并且持续火热，带来了新一轮的 IT 变革。云计算带给企业的创新能力和发展空间是不可想象的，我们所有人都正处于云计算大潮中。

云计算从狭义上讲，指 IT 基础设施的交付和使用模式，即通过网络以按需、易扩展的方式获取所需资源。广义上则指服务的交付和使用模式，通过网络以按需、易扩展的方式获取所需服务。提供资源的网络被形象地比喻成“云”，其计算能力通常是由分布式的大规模集群和虚拟化技术提供的。而“云”中的计算资源在用户看来是可以扩展，并且可以随时获取、按需使用的。

云计算彻底改变了人们对计算资源的使用方式，有一个形象的比喻说明了云计算革命

性的影响：“云”好比一个发电厂，互联网好比是输电线路，只不过这个发电厂对外提供的是 IT 服务，这种服务将通过互联网传输到千家万户。云计算实现了计算资源从单台发电机供电模式向电厂集中供电模式的转变。

业界根据云计算提供服务资源的类型将其划分为三大类：基础设施即服务（Infrastructure-as-a-Service，IaaS）、平台即服务（Platform-as-a-Service，Paas）和软件即服务（Software-as-a-Service，SaaS），如图 1-1 所示。

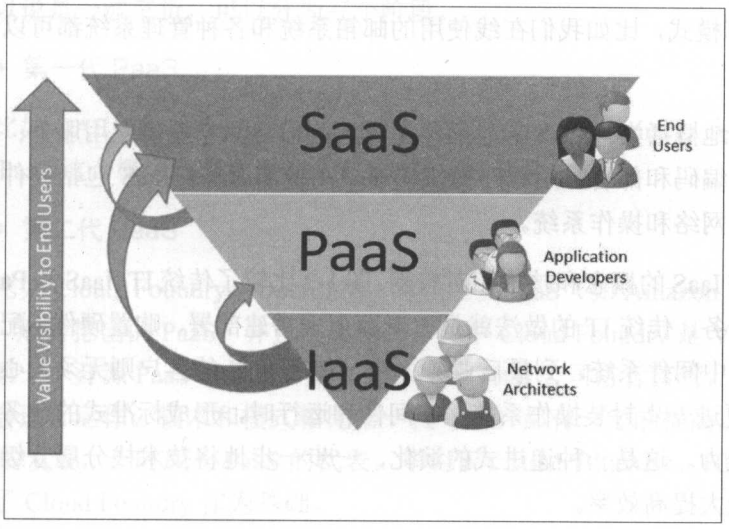


图 1-1　云计算的三层架构

- **基础设施即服务**

基础设施即服务（IaaS）通过虚拟化和分布式存储等技术，实现了对包括服务器、存储设备、网络设备等各种物理资源的抽象，从而形成了一个可扩展、可按需分配的虚拟资源池。IaaS 对外呈现的服务是各种基础设置，例如虚拟机、磁盘以及主机互联而成的网络，这些虚拟机中可以运行 Windows 系统，也可以运行 Linux 系统，在用户看来，它与一台真实的物理机是没有区别的。目前最具代表性的 IaaS 产品有 Amazon AWS，其提供了虚拟机 EC2 和云存储 S3 等服务。

- **平台即服务**

平台即服务（PaaS）为开发者提供了应用的开发环境和运行环境，将开发者从烦琐的 IT 环境管理中解放出来。自动化应用的部署和运维，使开发者能够集中精力于应用业务开

发，极大地提升了应用的开发效率。可以说，PaaS 主要面向的是软件专业人员，Google 的 GAE 是 PaaS 的鼻祖，而 Kubernetes 可以说是在 PaaS 的定义范畴内。

- **软件即服务**

软件即服务（SaaS）主要面向使用软件的终端用户。一般来说，SaaS 将软件功能以特定的接口形式发布，终端用户通过网络浏览器就可以使用软件功能。终端用户将只关注软件业务的使用，除此之外的工作，如软件的升级和云端实现，对终端用户来说都是透明的。SaaS 是应用最广的云计算模式，比如我们在线使用的邮箱系统和各种管理系统都可以认为是 SaaS 的范畴。

综上所述，可以简单地概括为：SaaS 通过网络运行，为最终用户提供应用服务；PaaS 是一套工具服务，可以为编码和部署应用程序提供快速、高效的服务；IaaS 包括硬件和软件，例如服务器、存储、网络和操作系统。

与 SaaS 相比，PaaS 和 IaaS 的概念和技术相对较新，图 1-2 比较了传统 IT、IaaS 和 PaaS。假设现在要上线一项新业务，传统 IT 的做法就是自下而上地搭建部署、购置硬件、配置网络、安装操作系统、部署中间件系统，到最后业务上线。使用 IaaS 的客户则无须关心操作系统以下的实现，Paas 更进一步封装操作系统、中间件和运行时，形成标准式的业务发布平台，提供智能化运维能力。这是一种递进式的演化，一步一步地将技术栈分层分级，将资源进行整合管理，可极大提高效率。

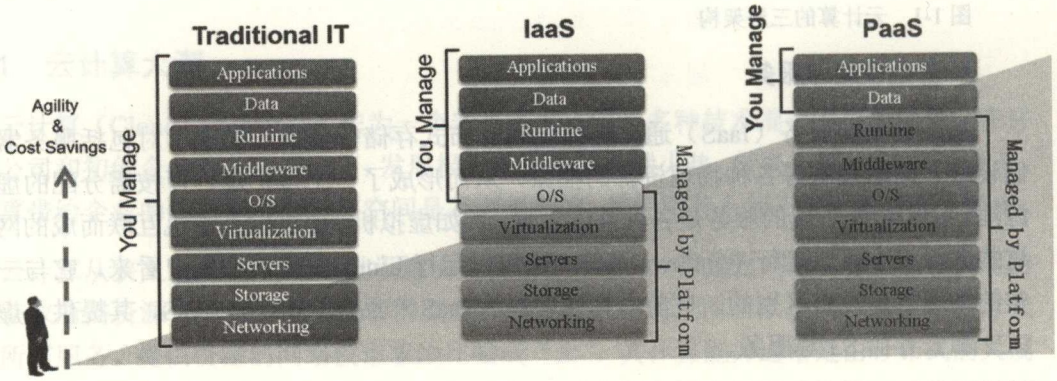


图 1-2　传统 IT、IaaS 和 PaaS 的比较

正是由于云计算的强大优势，越来越多的公司进入这波潮流中，形成了百家齐放的场面。在云计算的不同层次，在各个行业的不同领域，都涌现出一大批云计算产品，整个云

计算市场正在高速发展。

1.1.2 不温不火的 PaaS

在 SaaS 的成熟和 IaaS 的高速发展催生下，特别是在 Amazon、Google、Salesforce、Microsoft 等公司的推动下，PaaS 得到了长足的发展，越来越多的人开始谈论和关注 PaaS，包括运营商、互联网巨头、传统 IT 厂商、咨询和集成商、IT 技术媒体等。但是 PaaS 的发展可以说是一波三折，可以分为三个阶段。

- 第一代 PaaS

比如 GAE（Google App Engine）、SAE（Sina App Engine）。这是早期的 PaaS，当时并没有 PaaS 这个概念，现在看来是包含在 PaaS 范围内的。

- 第二代 PaaS

比如 Cloud Foundry、Openshift。这是各大 IaaS（如 Amazon AWS、OpenStack）流行之后，顺势推出的 PaaS，并且发展迅速。其中 Cloud Foundry 是 VMware 于 2011 年推出的业界第一个开源 PaaS 云平台，后来分拆出 Pivotal 公司进行接管，2014 创立 Cloud Foundry 基金会进行运作。技术和模式相比第一代 PaaS 都有一定的提高，在云计算大潮中引领了 PaaS 的发展，一时成为 PaaS 的代表。华为云、IBM BlueMix、HP Cloud 和 Dell 云服务都采用了 Cloud Foundry 作为基础。

但是这个阶段的 PaaS 不管是在市场份额，还是提升速度上都处于弱势，用户对 PaaS 的兴趣似乎也不大。同时，随着各种云服务之间界限的逐步模糊，一部分人甚至认为 PaaS 将最终消亡或成为 IaaS 或者 SaaS 的一个功能，PaaS 处于不温不火的尴尬位置。

- 第三代 PaaS

在 Docker 火爆之后，利用 Docker 的特性构建出许多 PaaS，比如 Kubernetes。这些 PaaS 更加灵活，更加适应企业，逐渐成为 PaaS 的主力。

1.1.3 Docker 的逆袭

Docker 是一种 Linux 容器工具集，它是为构建（Build）、交付（Ship）和运行（Run）分布式应用而设计的。作为 DotCloud 公司的开源项目，其首发版本的时间是 2013 年 3 月。该项目很快就受到欢迎，这也使得 DotCloud 公司将其品牌改为 Docker，并最终将其原有的

PaaS 业务出售而专注在 Docker 上，Docker 完成了华丽的逆袭。

Docker 设计理论来自集装箱，假设交付运行环境如同海运，操作系统如同一艘货轮，每一个在操作系统基础上运行的软件都如同一个集装箱，用户可以通过标准化手段自由组装运行环境，同时集装箱的内容可以由用户自定义，也可以由专业人员制造。这样，交付一个软件，就是一系列标准化组件的集合的交付，如同搭建乐高积木，用户只需选择合适的积木组合，并且在顶端署上自己的名字，最后这个标准化组件就是用户的应用。

基于这个理念，在技术实现上，Docker 利用容器（Container）来实现类似虚拟机的功能，从而利用更加节省的硬件资源提供给用户更多的计算资源。同虚拟机的方式不同，容器并不是一套硬件虚拟化方法，也无法归属到全虚拟化、部分虚拟化和半虚拟化中的任意一个，而是一个操作系统级虚拟化方法。

Docker 容器技术的优势有以下几点。

- **一次构建，到处运行**
  当将容器固化成镜像后，可以快速地加载到任何环境中部署运行。而构建出来的镜像打包了应用运行所需的程序、依赖和运行环境，这是一个完整可用的应用集装箱，在任何环境下都能保证环境的一致性。
- **容器的快速轻量**
  容器的启动、停止和销毁都是以秒或毫秒为单位的，并且相比传统的虚拟化技术，使用容器在 CPU、内存，网络 I/O 等资源上的性能损耗都有同样水平甚至更优的表现。
- **完整的生态链**
  容器技术并不是 Docker 首创，但是以往的容器实现只关注于如何运行，而 Docker 站在巨人的肩膀上进行了整合和创新，特别是 Docker 镜像的设计，完美地为容器从构建、交付到运行提供了完整的生态链支持。

Docker 1.0 在 2014 年 6 月发布，而且延续了之前每月发布一个版本的节奏。其 1.0 版本标志着 Docker 公司认为 Docker 平台已经足够成熟，并可以被应用到生产环境中。每月的版本更新显示出该项目正在快速发展，比如增加新的特性，解决发现的问题等。

Docker 的持续火热是有着坚实的基础来支撑的。Docker 吸引了业界众多知名大牌厂家的支持，其中包括 Amazon、Canonical、CenturyLink、Google、IBM、Microsoft、New Relic、Pivotal、Red Hat 和 VMware，这使得只要在有 Linux 的地方，Docker 就几乎随处可用。除

了这些大厂，许多初创企业也围绕着 Docker 来发展，或是将他们的发展方向和 Docker 更好地结合起来。所有这些合作伙伴都驱动着 Docker 核心项目和周边生态系统的快速发展。

更重要的是 Docker 的流行和标准化，激活了一直不温不火的 PaaS，随之而来的是各类 Micro-PaaS 的出现，Kubernetes 是其中最具代表性的一员。

1.2 Kubernetes 是什么

Kubernetes 是 Google 开源的容器集群管理系统。它构建在 Docker 技术之上，为容器化的应用提供资源调度、部署运行、服务发现、扩容缩容等一整套功能，本质上可看作是基于容器技术的 Micro-PaaS 平台，即第三代 PaaS 的代表性项目。

Google 从 2004 年起就已经开始使用容器技术了，于 2006 年发布了 Cgroup，而且内部开发了强大的集群资源管理平台 Borg 和 Omega，这些都已经广泛使用在 Google 的各个基础设施中，而 Kubernetes 的灵感来源于 Google 的内部 Borg 系统，更是吸收了包括 Omega 在内的容器管理器的经验和教训。

Kubernetes，古希腊语是舵手的意思，也是 Cyber 的词源，Kubernetes 利用 Google 在容器技术上的实践经验和技术积累，同时吸取 Docker 社区的最佳实践，已经成为云计算服务的舵手。

Kubernetes 有着如下的优秀特性。

- **强大的容器编排能力**

  Kubernetes 可以说是同 Docker 一起发展起来的，深度集成了 Docker，天然适应容器的特点，设计出强大的容器编排能力，比如容器组合、标签选择和服务发现等，可以满足企业级需求。

- **轻量级**

  Kubernetes 遵循微服务架构理论，整个系统划分出各个功能独立的组件，组件之间边界清晰，部署简单，可以轻易地运行在各种系统和环境中。同时，Kubernetes 中的许多功能都实现了插件化，可以非常方便地进行扩展和替换。

- **开放开源**

  Kubernetes 顺应了开放开源的趋势，吸引了大批开发者和公司参与其中，协同工作共同构建生态圈。同时，Kubernetes 同 OpenStack、Docker 等开源社区积极合作、共同发展，企业和个人都可以参与其中并获益。

1.3 Kubernetes 的发展历史

Kubernetes 自推出后迅速得到关注和参与，2015 年 7 月经过 400 多位贡献者一年的努力，多达 14,000 次代码提交，Google 正式对外发布了 Kubernetes v1.0，意味着这个开源容器编排系统可以正式在生产环境中使用。与此同时，谷歌联合 Linux 基金会及其他合作伙伴共同成立了 CNCF 基金会（Cloud Native Computing Foundation），并将 Kubernetes 作为首个编入 CNCF 基金会管理体系的开源项目，助力容器技术生态的发展进步。

Kubernetes 的发展里程碑如下所示。

- 2014 年 6 月：谷歌宣布 Kubernetes 开源。
- 2014 年 7 月：Microsoft、Red Hat、IBM、Docker、CoreOS、Mesosphere 和 Saltstack 加入 Kubernetes。
- 2014 年 8 月：Mesosphere 宣布将 Kubernetes 作为框架整合到 Mesosphere 生态系统中，用于 Docker 容器集群的调度、部署和管理。
- 2014 年 8 月：VMware 加入 Kubernetes 社区，Google 产品经理 Craig Mcluckie 公开表示，VMware 将会帮助 Kubernetes 实现利用虚拟化来保证物理主机安全的功能模式。
- 2014 年 11 月：HP 加入 Kubernetes 社区。
- 2014 年 11 月：Google 容器引擎 Alpha 启动，谷歌宣布 GCE 支持容器及服务，并以 Kubernetes 为构架。
- 2015 年 1 月：Google 和 Mirantis 及伙伴将 Kubernetes 引入 OpenStack，开发者可以在 OpenStack 上部署运行 Kubernetes 应用。
- 2015 年 4 月：Google 和 CoreOS 联合发布 Tectonic，它将 Kubernetes 和 CoreOS 软件栈整合在了一起。
- 2015 年 5 月：Intel 加入 Kubernetes 社区，宣布将合作加速 Tectonic 软件栈的发展进度。
- 2015 年 6 月：Google 容器引擎进入 beta 版。
- 2015 年 7 月：Google 正式加入 OpenStack 基金会，Kubernetes 产品经理 Craig McLuckie 宣布 Google 将成为 OpenStack 基金会的发起人之一，Google 将把他的容器计算的专家技术带入 OpenStack，以提高公有云和私有云的互用性。
- 2015 年 7 月：Kubernetes v1.0 正式发布。

1.4 Kubernetes 的核心概念

1.4.1 Pod

Pod 是若干相关容器的组合，Pod 包含的容器运行在同一台宿主机上，这些容器使用相同的网络命名空间、IP 地址和端口，相互之间能通过 localhost 来发现和通信。另外，这些容器还可共享一块存储卷空间。在 Kubernetes 中创建、调度和管理的最小单位是 Pod，而不是容器，Pod 通过提供更高层次的抽象，提供了更加灵活的部署和管理模式。

1.4.2 Replication Controller

Replication Controller 用来控制管理 Pod 副本（Replica，或者称为实例），Replication Controller 确保任何时候 Kubernetes 集群中有指定数量的 Pod 副本在运行。如果少于指定数量的 Pod 副本，Replication Controller 会启动新的 Pod 副本，反之会杀死多余的副本以保证数量不变。另外，Replication Controller 是弹性伸缩、滚动升级的实现核心。

1.4.3 Service

Service 是真实应用服务的抽象，定义了 Pod 的逻辑集合和访问这个 Pod 集合的策略。Service 将代理 Pod 对外表现为一个单一访问接口，外部不需要了解后端 Pod 如何运行，这给扩展和维护带来很多好处，提供了一套简化的服务代理和发现机制。

1.4.4 Label

Label 是用于区分 Pod、Service、Replication Controller 的 Key/Value 对，实际上，Kubernetes 中的任意 API 对象都可以通过 Label 进行标识。每个 API 对象可以有多个 Label，但是每个 Label 的 Key 只能对应一个 Value。Label 是 Service 和 Replication Controller 运行的基础，它们都通过 Label 来关联 Pod，相比于强绑定模型，这是一种非常好的松耦合关系。

1.4.5 Node

Kubernetes 属于主从分布式集群架构，Kubernetes Node（简称为 Node，早期版本叫作 Minion）运行并管理容器。Node 作为 Kubernetes 的操作单元，用来分配给 Pod（或者说容器）进行绑定，Pod 最终运行在 Node 上，Node 可以认为是 Pod 的宿主机。

第 2 章

Kubernetes 的架构和部署

Kubernetes 遵循微服务架构理论，整个系统划分出各个功能独立的组件，组件之间边界清晰，部署简单，可以轻易地运行在各种系统和环境中。本章将首先介绍 Kubernetes 的架构和各个组件的功能，然后详细说明如何从头开始部署一套完整的 Kubernetes 运行环境，包括 Kubernetes 提供的各种扩展插件（Cluster Add-on）的安装方式。

2.1 Kubernetes 的架构和组件

Kubernetes 属于主从分布式架构，节点在角色上分为 Master 和 Node，如图 2-1 所示。

Kubernetes 使用 Etcd 作为存储中间件，Etcd 是一个高可用的键值存储系统，灵感来自于 ZooKeeper 和 Doozer，通过 Raft 一致性算法处理日志复制以保证强一致性。Kubernetes 使用 Etcd 作为系统的配置存储中心，Kubernetes 中的重要数据都是持久化在 Etcd 中的，这使得 Kubernetes 架构的各个组件属于无状态，可以更简单地实施分布式集群部署。

Kubernetes Master 作为控制节点，调度管理整个系统，包含以下组件。

- Kubernetes API Server：作为 Kubernetes 系统的入口，其封装了核心对象的增删改查操作，以 REST API 接口方式提供给外部客户和内部组件调用。它维护的 REST 对象将持久化到 Etcd 中。
- Kubernetes Scheduler：负责集群的资源调度，为新建的 Pod 分配机器。这部分工作分出来变成一个组件，意味着可以很方便地替换成其他的调度器。

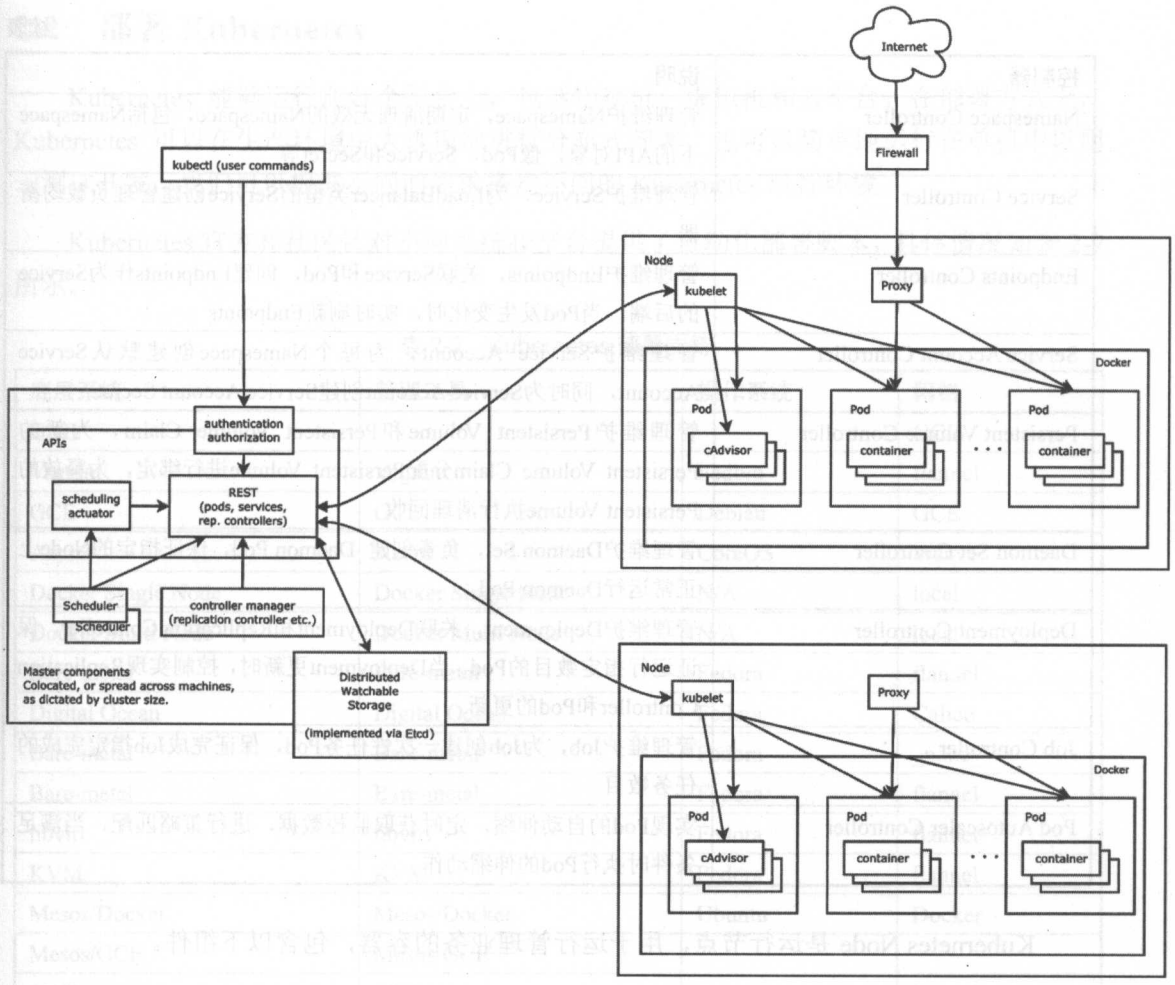


图 2-1 Kubernetes 的架构

- Kubernetes Controller Manager：负责执行各种控制器，目前已经实现很多控制器来保证 Kubernetes 的正常运行，主要包含的控制器如表 2-1 所示。

表 2-1 Kubernetes 控制器

控制器	说明
Replication Controller	管理维护Replication Controller，关联Replication Controller和Pod，保证Replication Controller定义的副本数量与实际运行Pod的数量是一致的
Node Controller	管理维护Node，定期检查Node的健康状态，标识出失效的Node

续表

控制器	说明
Namespace Controller	管理维护Namespace，定期清理无效的Namespace，包括Namespace下的API对象，像Pod、Service和Secret等
Service Controller	管理维护Service，为LoadBalancer类型的Service创建管理负载均衡器
Endpoints Controller	管理维护Endpoints，关联Service和Pod，创建Endpoints作为Service的后端，当Pod发生变化时，实时刷新Endpoints
Service Account Controller	管理维护Service Account，为每个Namespace创建默认Service Account，同时为Service Account创建Service Account Secret
Persistent Volume Controller	管理维护Persistent Volume和Persistent Volume Claim，为新的Persistent Volume Claim分配Persistent Volume进行绑定，为释放的Persistent Volume执行清理回收
Daemon Set Controller	管理维护Daemon Set，负责创建 Daemon Pod，保证指定的Node上正常运行Daemon Pod
Deployment Controller	管理维护Deployment，关联Deployment和Replication Controller，保证运行指定数目的Pod。当Deployment更新时，控制实现Replication Controller和Pod的更新
Job Controller	管理维护Job，为Job创建一次性任务Pod，保证完成Job指定完成的任务数目
Pod Autoscaler Controller	实现Pod的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执行Pod的伸缩动作

Kubernetes Node 是运行节点，用于运行管理业务的容器，包含以下组件。

- Kubelet：负责管控容器，Kubelet 会从 Kubernetes API Server 接收 Pod 的创建请求，启动和停止容器，监控容器运行状态并汇报给 Kubernetes API Server。
- Kubernetes Proxy：负责为 Pod 创建代理服务，Kubernetes Proxy 会从 Kubernetes API Server 获取所有的 Service，并根据 Service 信息创建代理服务，实现 Service 到 Pod 的请求路由和转发，从而实现 Kubernetes 层级的虚拟转发网络。
- Docker：Kubernetes Node 是容器运行节点，需要运行 Docker 服务，目前 Kubernetes 也支持 Rocket，这是一款 CoreOS 开发的类 Docker 的开源容器引擎，本书只说明 Docker。

2.2 部署 Kubernetes

Kubernetes 能够运行在各个平台上，包括物理机、虚拟机和云平台。在部署方式上，Kubernetes 可以在生产环境中大规模地进行分布式部署，也可以简单地运行在单机中以用于测试开发，我们可以根据不同的需求搭建不同的 Kubernetes 运行环境。

Kubernetes 官方和社区针对不同系统和平台提供了自动化部署脚本，具体情况如表 2-2 所示。

表 2-2 Kubernetes 部署支持

底层系统	管理工具	操作系统	网络
GKE			GCE
Vagrant	Vagrant	Fedora	flannel
GCE	GCE	Debian	GCE
Azure	Azure	CoreOS	Weave
Docker Single Node	Docker Single Node	N/A	local
Docker Multi Node	Docker Multi Node	N/A	local
Bare-metal	Bare-metal	Fedora	flannel
Digital Ocean	Digital Ocean	Fedora	Calico
Bare-metal	Bare-metal	Fedora	_none_
Bare-metal	Bare-metal	Fedora	flannel
libvirt	libvirt	Fedora	flannel
KVM	KVM	Fedora	flannel
Mesos/Docker	Mesos/Docker	Ubuntu	Docker
Mesos/GCE	Mesos/GCE		
AWS	AWS	CoreOS	flannel
GCE	GCE	CoreOS	flannel
Vagrant	Vagrant	CoreOS	flannel
Bare-metal (Offline)	Bare-metal (Offline)	CoreOS	flannel
Bare-metal	Bare-metal	CoreOS	Calico
CloudStack	CloudStack	CoreOS	flannel
VMware	VMware	Debian	OVS
Bare-metal	Bare-metal	CentOS	
AWS	AWS	Ubuntu	flannel

续表

底层系统	管理工具	操作系统	网络
OpenStack/HPCloud	OpenStack/HPCloud	Ubuntu	flannel
Joyent	Joyent	Ubuntu	flannel
AWS	AWS	Ubuntu	OVS
Azure	Azure	Ubuntu	OpenVPN
Bare-metal	Bare-metal	Ubuntu	Calico
Bare-metal	Bare-metal	Ubuntu	flannel
Local	Local		
libvirt/KVM	libvirt/KVM	CoreOS	libvirt/KVM
oVirt	oVirt		
Rackspace	Rackspace	CoreOS	flannel

接下来我们将详细说明Kubernetes的部署步骤，同时也会涉及Kubernetes的一些概念，可以进一步帮助读者理解 Kubernetes 的架构和原理。

2.2.1 环境准备

Kubernetes 是一个分布式架构，可灵活地进行安装部署，可以部署在单机，也可以分布式部署。机器可以是物理机，也可以是虚拟机，但是需要运行 Linux（x86_64）操作系统，至少 1 核 CPU 和 1GB 内存。

本书准备了 4 台虚拟机（CentOS 7.0 系统）用于部署 Kubernetes 运行环境，包括一个 Etcd、一个 Kubernetes Master 和三个 Kubernetes Node，如表 2-3 所示。

表 2-3 Kubernetes 运行环境

节点	主机名	IP
Etcd	etcd	192.168.3.145
Kubernetes Master	kube-master	192.168.3.146
Kubernetes Node 1	kube-node-1	192.168.3.147
Kubernetes Node 2	kube-node-2	192.168.3.148
Kubernetes Node 3	kube-node-3	192.168.3.149

使用的各种软件版本信息如表 2-4 所示。

表 2-4 软件版本

软件	版本
Kubernetes	1.1.1
Docker	1.7.1
Etcd	2.2.0
Flannel	0.5.1
Open vSwitch	2.3.1

提示

本书后续的实践操作都在此套环境中运行。

2.2.2 运行 Etcd

Kubernetes 依赖于 Etcd，需要先部署 Etcd，可以从 Github 上下载指定版本的 Etcd 发布包：

```
$ wget https://github.com/coreos/etcd/releases/download/v2.2.0/etcd-v2.2.0-linux-amd64.tar.gz
$ tar xzvf etcd-v2.2.0-linux-amd64.tar.gz
$ cd etcd-v2.2.0-linux-amd64
$ cp etcd /usr/bin/etcd
$ cp etcdctl /usr/bin/etcdctl
```

运行 Etcd：

```
$ etcd -name etcd \
-data-dir /var/lib/etcd \
-listen-client-urls http://0.0.0.0:2379, http://0.0.0.0:4001 \
-advertise-client-urls http://0.0.0.0:2379, http://0.0.0.0:4001 \
>> /var/log/etcd.log 2>&1 &
```

Etcd 运行后可以查询其健康状态：

```
$ etcdctl -C http://etcd:4001 cluster-health
member ce2a822cea30bfca is healthy: got healthy result from http://etcd:4001
cluster is healthy
```

2.2.3 获取 Kubernetes 发布包

Kubernetes 发布包可以从 Github 上下载，本书使用的是 Kubernetes V1.1.1 版本：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$tar zxvf kubernetes.tar.gz
```

Kubernetes 发布包中包含如下内容。

- cluster：Kubernetes 自动化部署脚本。
- contrib：Kubernetes 非必需程序。
- examples：Kubernetes 示例配置文件。
- docs：Kubernetes 文档。
- platforms：Kubernetes 的命令行工具 kubectl。
- server：Kubernetes 组件。
- third_party：第三方程序。

Kubernetes 发布包中的 server/kubernetes-server-linux-amd64.tar.gz 包含各个组件的可执行程序，将这些可执行程序复制到 Linux 系统目录/use/bin/下：

```
$ cd kubernetes/server
$ tar zxvf kubernetes-server-linux-amd64.tar.gz
$ cd kubernetes/server/bin/
$ find ./ -perm 755 | xargs -i cp {} /usr/bin/
```

2.2.4 运行 Kubernetes Master 组件

在 Kubernetes Master 上需要运行以下组件：

- Kubernetes API Server
- Kubernetes Controller Manager
- Kubernetes Scheduler
- Kubernetes Proxy（可选）

Kubernetes API Server

运行 Kubernetes API Server：

```
$ kube-apiserver \
--logtostderr=true --v=0 \
```

```
--etcd_servers=http://etcd:4001 \
--insecure-bind-address=0.0.0.0 --insecure-port=8080 \
--service-cluster-ip-range=10.254.0.0/16  \
>> /var/log/kube-apiserver.log 2>&1 &
```

Kubernetes Controller Manager

运行 Kubernetes Controller Manager：

```
$ kube-controller-manager \
--logtostderr=true --v=0 \
--master=http://kube-master:8080  \
>> /var/log/kube-controller-manager.log 2>&1 &
```

Kubernetes Scheduler

运行 Kubernetes Scheduler：

```
$ kube-scheduler \
--logtostderr=true --v=0 \
--master=http://kube-master:8080  \
>> /var/log/kube-scheduler.log 2>&1 &
```

Kubernetes Proxy

运行 Kubernetes Proxy：

```
$ kube-proxy \
--logtostderr=true --v=0 \
--api_servers=http://kube-master:8080  \
>> /var/log/kube-proxy.log 2>&1 &
```

2.2.5 运行 Kubernetes Node 组件

Kubernetes Node 上需要运行以下组件：

- Docker
- Kubelet
- Kubernetes Proxy

Docker

Docker 官方社区提供各个平台的安装方法（可参考 http://docs.docker.com/installation/），很多 Linux 发行版都陆续对 Docker 进行了支持，可以使用以下方式快速安装 Docker：

```
$ curl -sSL https://get.docker.com/ | sh
```

提示

1. Docker 要求 Linux 64 系统，并且内核至少 3.10 版本以上。
2. 不同的 Kubernetes 版本对 Docker 的版本要求可能会有不同，总体上我们建议使用最新稳定的 Docker 版本，如果 Kubelet 发现 Docker 版本过低，在创建 Pod 的时候会失败并发出告警日志。

启动 Docker：

```
$ docker -d \
-H unix:///var/run/docker.sock -H 0.0.0.0:2375 \
>> /var/log/docker.log 2>&1 &
```

Kubelet

运行 Kubelet：

```
$ kubelet \
--logtostderr=true --v=0 \
--config=/etc/kubernetes/manifests \
--address=0.0.0.0 \
--api-servers=http://kube-master:8080  \
>> /var/log/kubelet.log 2>&1 &
```

Kubernetes Proxy

运行 Kubelet Proxy：

```
$ kube-proxy \
--logtostderr=true --v=0 \
--api_servers=http://kube-master:8080  \
>> /var/log/kube-proxy.log 2>&1 &
```

2.2.6 查询 Kubernetes 的健康状态

在部署运行各个组件以后，可以通过 Kubernetes 命令行 kubectl 查询 Kubernetes Master

各组件的健康状态：

```
$ kubectl -s http://kube-master:8080 get componentstatus
NAME                 STATUS    MESSAGE                         ERROR
controller-manager   Healthy   ok                              nil
scheduler            Healthy   ok                              nil
etcd-0               Healthy   {"health": "true"}              nil
```

以及 Kubernetes Node 的健康状态：

```
$ kubectl -s http://kube-master:8080 get node
NAME          LABELS                                STATUS    AGE
kube-node-1   kubernetes.io/hostname=kube-node-1    Ready     19m
kube-node-2   kubernetes.io/hostname=kube-node-2    Ready     18m
kube-node-3   kubernetes.io/hostname=kube-node-3    Ready     18m
```

2.2.7 创建 Kubernetes 覆盖网络

Kubernetes 的网络模型要求每一个 Pod 都拥有一个扁平化共享网络命名空间的 IP，称为 PodIP，Pod 能够直接通过 PodIP 跨网络与其他物理机和 Pod 进行通信。要实现 Kubernetes 的网络模型，需要在 Kubernetes 集群中创建一个覆盖网络（Overlay Network），联通各个节点，目前可以通过第三方网络插件来创建覆盖网络，比如 Flannel 和 Open vSwitch。

提示

本书中用到的 Kubernetes 运行环境主要是使用 Flannel 实现容器覆盖网络的。

Flannel

Flannel 是 CoreOS 团队设计开发的一个覆盖网络工具，可以从 Github 上下载指定版本的 Flannel 发布包：

```
$ wget https://github.com/coreos/flannel/releases/download/v0.5.4/flannel-0.5.4-linux-amd64.tar.gz
$ tar xzvf flannel-0.5.4-linux-amd64.tar.gz
$ cd flannel-0.5.4
$ cp flanneld /usr/bin
```

Flannel 使用 Etcd 进行配置，来保证多个 Flannel 实例之间的配置一致性，所以需要首先在 Etcd 上进行配置：

```
$ etcdctl -C http://etcd:4001 \
```

```
set /coreos.com/network/config '{ "Network": "10.0.0.0/16" }'
```

在 Kubernetes 的所有节点上运行 Flannel:

```
$ flanneld -etcd-endpoints=http://etcd:4001 \
>> /var/log/flanneld.log 2>&1 &
```

Flannel 会为不同 Node 的 Docker 网桥配置不同的 IP 网段以保证 Docker 容器的 IP 在集群内唯一，所以 Flannel 会重新配置 Docker 网桥，需要先删除原先创建的 Docker 网桥：

```
$ iptables -t nat -F
$ ifconfig docker0 down
$ brctl delbr docker0
```

Flannel 运行后会生成一个文件 subnet.env，其中包含规划好的 Docker 网桥网段，根据其中的属性重新启动 Docker：

```
$ source /run/flannel/subnet.env
$ docker -d \
-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 \
--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} \
>> /var/log/docker.log 2>&1 &
```

Open vSwitch

Open vSwitch 是一个高质量的、多层虚拟交换机，使用开源 Apache 2.0 许可协议，由 Nicira Networks 开发。它的目的是让大规模网络自动化可以通过编程扩展，同时仍然支持标准的管理接口和协议。Open vSwitch 是一项非常重要的 SDN 技术，可以实现 Kubernetes 中的覆盖网络。

为保证 Docker 容器的 IP 在集群内唯一，不同 Kubernetes Node 的 Docker 网桥配置成不同的 IP 网段，需要进行规划，如表 2-5 所示。

表 2-5 Kubernetes Node 的 Docker 网桥规划

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.246.0.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.246.1.1/24
Kubernetes Node 3	kube-node-3	192.168.3.149	10.246.2.1/24

安装运行 Open vSwitch 服务以后，可以下载一个工具来协助创建 Open vSwitch 网络：

```
$ wget https://raw.githubusercontent.com/wulonghui/docker-net-tools/master/k8s-ovs-ctl
$ chmod 0750 k8s-ovs-ctl
$ mv k8s-ovs-ctl /usr/bin/
```

在 Kubernetes Node 上配置~ /k8s-ovs.env：

- Kubernetes Node 1

```
DOCKER_BRIDGE=docker0
CONTAINER_ADDR=10.246.0.1
CONTAINER_NETMASK=255.255.255.0
CONTAINER_SUBNET=10.246.0.0/16

OVS_SWITCH=obr0
TUNNEL_BASE=gre
DOCKER_OVS_TUN=tun0

LOCAL_IP=192.168.3.147
NODE_IPS=(192.168.3.147  192.168.3.148  192.168.3.149)
CONTAINER_SUBNETS=(10.246.0.1/24  10.246.1.1/24  10.246.2.1/24)
```

- Kubernetes Node 2

```
DOCKER_BRIDGE=docker0
CONTAINER_ADDR=10.246.1.1
CONTAINER_NETMASK=255.255.255.0
CONTAINER_SUBNET=10.246.0.0/16

OVS_SWITCH=obr0
TUNNEL_BASE=gre
DOCKER_OVS_TUN=tun0

LOCAL_IP=192.168.3.148
NODE_IPS=(192.168.3.147  192.168.3.148  192.168.3.149)
CONTAINER_SUBNETS=(10.246.0.1/24  10.246.1.1/24  10.246.2.1/24)
```

- Kubernetes Node 3

```
DOCKER_BRIDGE=docker0
CONTAINER_ADDR=10.246.2.1
CONTAINER_NETMASK=255.255.255.0
CONTAINER_SUBNET=10.246.0.0/16
```

```
OVS_SWITCH=obr0
TUNNEL_BASE=gre
DOCKER_OVS_TUN=tun0

LOCAL_IP=192.168.3.149
NODE_IPS=(192.168.3.147  192.168.3.148  192.168.3.149)
CONTAINER_SUBNETS=(10.246.0.1/24  10.246.1.1/24  10.246.2.1/24)
```

配置完成后，先删除原先创建的 Docker 网桥：

```
$ iptables -t nat -F
$ ifconfig docker0 down
$ brctl delbr docker0
```

然后使用 k8s-ovs-ctl 创建 Open vSwitch 网络：

```
$ k8s-ovs-ctl setup
```

最后重新运行 Docker：

```
$ docker -d \
-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 \
--bridge=docker0 \
>> /var/log/docker.log 2>&1 &
```

2.3 安装 Kubernetes 扩展插件

Kubernetes 中提供了许多平台扩展插件（Cluster Add-on），包含在 Kubernetes 发布包中，可以在 Kubernetes 上进行安装部署，目前包含以下扩展插件：

- Cluster DNS
- Cluster Monitoring
- Cluster Logging
- Kube UI

提示

Kubernetes 平台扩展插件并不是必需的，初次部署可以跳过此章节，待后续阅读中有需要再进行安装。

2.3.1 安装 Cluster DNS

Cluster DNS 扩展插件用于支持 Kubernetes 的服务发现机制，Cluster DNS 主要包含如下几项。

- SkyDNS：提供 DNS 解析服务。
- Etcd：用于 SkyDNS 的存储。
- Kube2sky：监听 Kubernetes，当有新的 Service 创建时，生成相应记录到 SkyDNS。

下载 Kubernetes 发布包，Cluster DNS 扩展插件在 Kubernetes 发布包的 cluster/addons/dns 目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/dns
```

通过环境变量配置参数：

```
$ export DNS_SERVER_IP="10.254.10.2"
$ export DNS_DOMAIN="cluster.local"
$ export DNS_REPLICAS=1
```

设置 Cluster DNS Server 的 IP 为 10.254.10.2，Cluster DNS 的本地域为 cluster.local，另外需要配置到 Kubelet 的启动参数中：

```
--cluster-dns=10.254.10.2
--cluster-domain=cluster.local
```

Kubernetes 发布包 cluster/addons/dns 目录下的 skydns-rc.yaml.in 和 skydns-svc.yaml.in 是两个模板文件，通过设置的环境变量修改其中相应的属性值，生成 Replication Controller 和 Service 的定义文件：

```
$ sed -e "s/{{ pillar\['dns_replicas'\] }}/${DNS_REPLICAS}/g;s/{{ pillar\['dns_
domain'\]}}/${DNS_DOMAIN}/g;" \skydns-rc.yaml.in > skydns-rc.yaml
$ sed -e "s/{{ pillar\['dns_server'\] }}/${DNS_SERVER_IP}/g" skydns-svc.yaml.in >
skydns-svc.yaml
```

Cluster DNS Replication Controller 的定义文件 skydns-rc.yaml 如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v9
```

```
    namespace: kube-system
    labels:
      k8s-app: kube-dns
      version: v9
      kubernetes.io/cluster-service: "true"
  spec:
    replicas: 1
    selector:
      k8s-app: kube-dns
      version: v9
    template:
      metadata:
        labels:
          k8s-app: kube-dns
          version: v9
          kubernetes.io/cluster-service: "true"
      spec:
        containers:
        - name: etcd
          image: gcr.io/google_containers/etcd:2.0.9
          resources:
            limits:
              cpu: 100m
              memory: 50Mi
          command:
          - /usr/local/bin/etcd
          - -data-dir
          - /var/etcd/data
          - -listen-client-urls
          - http://127.0.0.1:2379,http://127.0.0.1:4001
          - -advertise-client-urls
          - http://127.0.0.1:2379,http://127.0.0.1:4001
          - -initial-cluster-token
          - skydns-etcd
          volumeMounts:
          - name: etcd-storage
            mountPath: /var/etcd/data
```

```
- name: kube2sky
  image: gcr.io/google_containers/kube2sky:1.11
  resources:
    limits:
      cpu: 100m
      memory: 50Mi
  args:
  # command = "/kube2sky"
  - -domain=cluster.local
  - -kube_master_url=http://kube-master:8080
- name: skydns
  image: gcr.io/google_containers/skydns:2015-10-13-8c72f8c
  resources:
    limits:
      cpu: 100m
      memory: 50Mi
  args:
  # command = "/skydns"
  - -machines=http://127.0.0.1:4001
  - -addr=0.0.0.0:53
  - -ns-rotate=false
  - -domain=cluster.local.
  ports:
  - containerPort: 53
    name: dns
    protocol: UDP
  - containerPort: 53
    name: dns-tcp
    protocol: TCP
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
      scheme: HTTP
    initialDelaySeconds: 30
    timeoutSeconds: 5
  readinessProbe:
```

```
      httpGet:
        path: /healthz
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 1
      timeoutSeconds: 5
  - name: healthz
    image: gcr.io/google_containers/exechealthz:1.0
    resources:
      limits:
        cpu: 10m
        memory: 20Mi
    args:
    - -cmd=nslookup kubernetes.default.svc.cluster.local localhost >/dev/null
    - -port=8080
    ports:
    - containerPort: 8080
      protocol: TCP
  volumes:
  - name: etcd-storage
    emptyDir: {}
  dnsPolicy: Default  # Don't use cluster DNS.
```

提示

kube2sky 需要 Service Account 来调用 Kubernetes API，如果 Kubernetes 没有开启 Service Account，可以显式指定 Kubernetes API 的 URL，在 skydns-rc.yaml 中设置 Kube2sky 的启动参数如下所示。

```
args:
  # command = "/kube2sky"
  - -domain=cluster.local
  - -kube_master_url=http://kube-master:8080
```

通过定义文件创建 Cluster DNS Replication Controller：

```
$ kubectl create -f skydns-rc.yaml
replicationcontroller "kube-dns-v9" created
```

Cluster DNS Replication Controller 创建运行 Cluster DNS Pod：

```
$ kubectl get pod --selector k8s-app=kube-dns --namespace=kube-system
NAME                READY     STATUS    RESTARTS AGE
kube-dns-v9-qdck6 4/4       Running   0        2m
```

Cluster DNS Service 的定义文件 skydns-svc.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP:  10.254.10.2
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

通过定义文件创建 Cluster DNS Service：

```
$ kubectl create -f skydns-svc.yaml
service "kube-dns" created
```

然后可以查询 Cluster DNS Service：

```
$ kubectl get service -l k8s-app=kube-dns --namespace=kube-system
NAME       CLUSTER_IP    EXTERNAL_IP   PORT(S)         SELECTOR           AGE
kube-dns   10.254.10.2   <none>        53/UDP,53/TCP   k8s-app=kube-dns   47s
```

Cluster DNS 部署完成后，可以创建 Pod 进行验证：

```
$ kubectl exec my-pod -- nslookup kubernetes.default.cluster.local
```

```
Server:    10.254.10.2
Address 1: 10.254.10.2

Name:      kubernetes.default.cluster.local
Address 1: 10.254.0.1
```

2.3.2 安装 Cluster Monitoring

Kubernetes 提供了 Cluster Monitoring 作为平台监控支持，Cluster Monitoring 的主体是 Heapster，一个容器集群的监控收集工具，将收集 Kubernetes 运行平台的监控数据，支持导入到其他第三方系统，比如 InfluxDB 和 GCE。

下载 Kubernetes 发布包，Cluster Monitoring 扩展插件在 Kubernetes 发布包的 cluster/addons/cluster-monitoring 目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/cluster-monitoring
```

在 Kubernetes 发布包的 cluster/addons/cluster-monitoring 目录中包含以下子目录，每个子目录中存放着不同的定义文件。

- standalone：部署独立运行的 Heapster。
- influxdb：部署 Heapster 对接 InfluxDB。
- google：部署 Heapster 对接 GCE。
- googleinfluxdb：部署 Heapster 对接 GCE 和 InfluxDB。

我们将部署 Heapster 对接 InfluxDB，InfluxDB 是一个开源分布式时序、事件和指标数据库，同时，InfluxDB 集成 Grafana 提供图表展示功能。

部署 InfluxDB 和 Grafana

Influxdb&Grafana Replication Controller 的定义文件 influxdb/influxdb-grafana-controller.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: monitoring-influxdb-grafana-v2
  namespace: kube-system
```

```
  labels:
    k8s-app: influxGrafana
    version: v2
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: influxGrafana
    version: v2
  template:
    metadata:
      labels:
        k8s-app: influxGrafana
        version: v2
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - image: gcr.io/google_containers/heapster_influxdb:v0.4
          name: influxdb
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
          ports:
            - containerPort: 8083
              hostPort: 8083
            - containerPort: 8086
              hostPort: 8086
          volumeMounts:
          - name: influxdb-persistent-storage
            mountPath: /data
        - image: beta.gcr.io/google_containers/heapster_grafana:v2.1.1
          name: grafana
          env:
          resources:
            limits:
              cpu: 100m
              memory: 100Mi
          env:
```

```
        # This variable is required to setup templates in Grafana.
        - name: INFLUXDB_SERVICE_URL
          value: http://monitoring-influxdb:8086
          # The following env variables are required to make Grafana accessible via
          # the kubernetes api-server proxy. On production clusters, we recommend
          # removing these env variables,setup auth for grafana,and expose the grafana
          # service using a LoadBalancer or a public IP.
          #- name: GF_AUTH_BASIC_ENABLED
          # value: "false"
          #- name: GF_AUTH_ANONYMOUS_ENABLED
          # value: "true"
          #- name: GF_AUTH_ANONYMOUS_ORG_ROLE
          # value: Admin
          #- name: GF_SERVER_ROOT_URL
          # alue: /api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
        volumeMounts:
        - name: grafana-persistent-storage
          mountPath: /var
      volumes:
      - name: influxdb-persistent-storage
        emptyDir: {}
      - name: grafana-persistent-storage
        emptyDir: {}
```

提示

influxdb/influxdb-grafana-controller.yaml 中注释掉 Grafana 容器的环境变量：

```
# The following env variables are required to make Grafana accessible via
# the kubernetes api-server proxy. On production clusters, we recommend
# removing these env variables,setup auth for grafana,and expose the grafana
# service using a LoadBalancer or a public IP.
#- name: GF_AUTH_BASIC_ENABLED
# value: "false"
#- name: GF_AUTH_ANONYMOUS_ENABLED
# value: "true"
#- name: GF_AUTH_ANONYMOUS_ORG_ROLE
# value: Admin
#- name: GF_SERVER_ROOT_URL
# alue: /api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
```

通过定义文件创建 Influxdb&Grafana Replication Controller：

```
$ kubectl create -f influxdb/influxdb-grafana-controller.yaml
replicationcontroller "monitoring-influxdb-grafana-v2" created
```

Influxdb&Grafana Replication Controller 创建运行 Influxdb&Grafana Pod：

```
$ kubectl get pod --selector k8s-app=influxGrafana --namespace=kube-system --output wide
NAME                                  READY     STATUS    RESTARTS   AGE       NODE
monitoring-influxdb-grafana-v2-v5e4y  2/2       Running   0          10m       kube-node-2
```

在 Pod 的查询信息中，属性 NODE 显示 Pod 运行在 Node kube-node-2 上，因为 Pod 中为 Influxdb 容器设置了端口映射规则（8083:8083），于是便可以通过 Node kube-node-2 访问到 Influxdb 的 Web 界面，访问地址是 http://kube-node-2:8083，如图 2-2 所示。

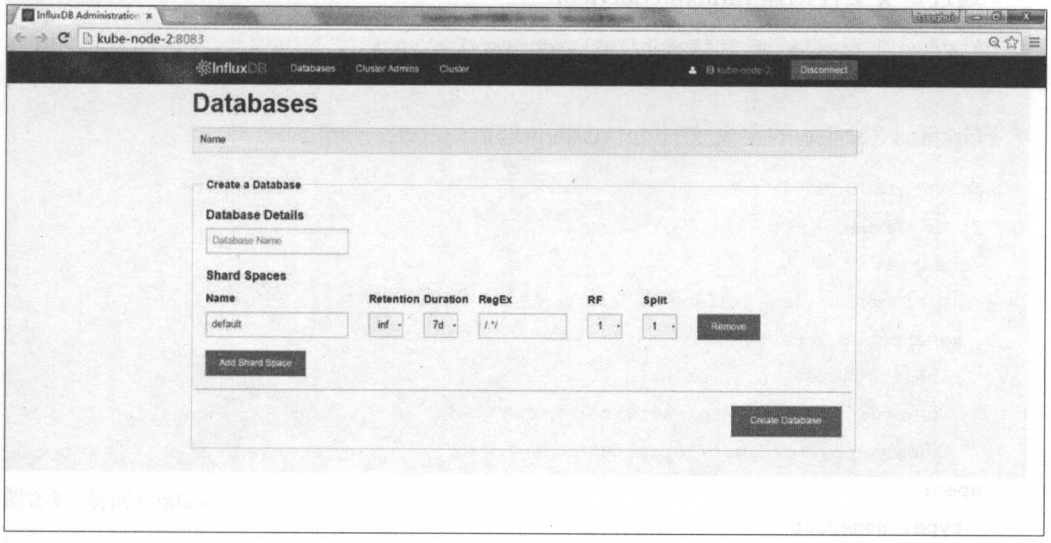


图 2-2 访问 Influxdb Web 界面

Influxdb Service 的定义文件 influxdb/influxdb-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: monitoring-influxdb
  namespace: kube-system
  labels:
```

```
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "InfluxDB"
spec:
  ports:
    - name: http
      port: 8083
      targetPort: 8083
    - name: api
      port: 8086
      targetPort: 8086
  selector:
    k8s-app: influxGrafana
```

通过定义文件创建 Influxdb Service：

```
$ kubectl create -f influxdb/influxdb-service.yaml
service "monitoring-influxdb" created
```

Grafana Service 的定义文件 influxdb/grafana-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: monitoring-grafana
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Grafana"
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 3000
  selector:
    k8s-app: influxGrafana
```

其中设置 Grafana Service 的类型为 NodePort，这样一来，Kubernetes 会为 Service 创建一个端口 NodePort，通过[Kubernetes Node IP]:[NodePort]就可以访问到 Service。

通过定义文件创建 Grafana Service：

```
$ kubectl create -f influxdb/grafana-service.yaml
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet,  you may
need to set up firewall rules for the service port(s) (tcp:32075) to serve traffic.

See http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md for more
details.
service "monitoring-grafana" created
```

从 Grafana Service 创建成功的提示中可以知道 Kubernetes 分配的 NodePort 是 tcp:32075，那么通过 http://kube-node-2:32075 可以访问 Grafana，如图 2-3 所示。

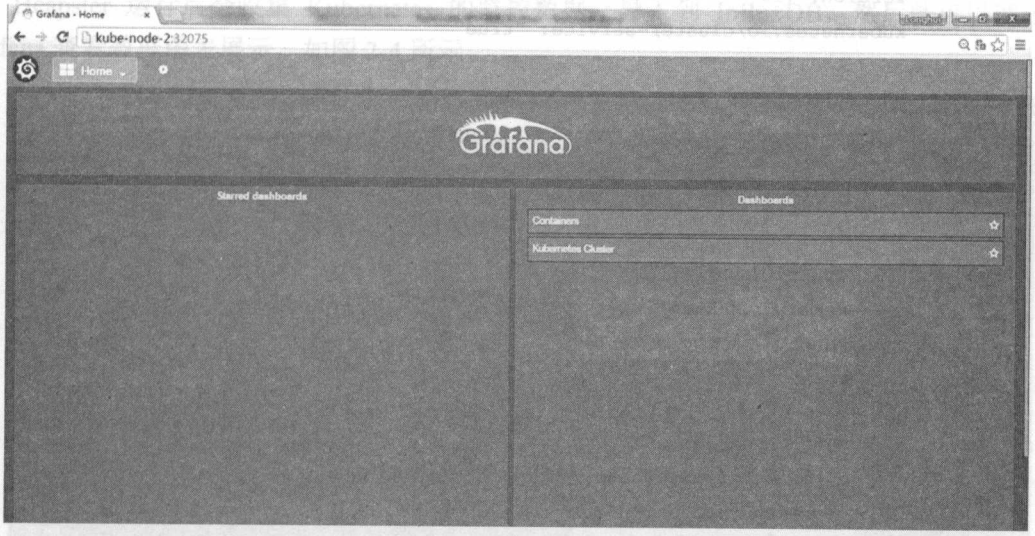


图 2-3　访问 Grafana

部署 Heapster

Heapster Replication Controller 的定义文件 influxdb/heapster-controller.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: heapster-v10
  namespace: kube-system
  labels:
```

```
        k8s-app: heapster
        version: v10
        kubernetes.io/cluster-service: "true"
    spec:
      replicas: 1
      selector:
        k8s-app: heapster
        version: v10
      template:
        metadata:
          labels:
            k8s-app: heapster
            version: v10
            kubernetes.io/cluster-service: "true"
        spec:
          containers:
            - image: gcr.io/google_containers/heapster:v0.18.2
              name: heapster
              resources:
                limits:
                  cpu: 100m
                  memory: 300Mi
              command:
                - /heapster
                - --source=kubernetes:http://kube-master:8080?inClusterConfig=
false&useServiceAccount=false
                - --sink=influxdb:http://monitoring-influxdb:8086
                - --stats_resolution=30s
                - --sink_frequency=1m
```

提示

Heapster 需要 Service Account 来调用 Kubernetes API，如果 Kubernetes 没有开启 Service Account，可以显式指定 Kubernetes API 的 URL，可在 skydns-rc.yaml 中设置 Heapster 的启动参数。

```
    command:
      - /heapster
      - --source=kubernetes:http://kube-master:8080?inClusterConfig=false&useServiceAccount=
false
```

```
- --sink=influxdb:http://monitoring-influxdb:8086
- --stats_resolution=30s
- --sink_frequency=1m
```

通过定义文件创建 Heapster Replication Controller：

```
$ kubectl create -f influxdb/heapster-controller.yaml
replicationcontroller "heapster-v10" created
```

Heapster Replication Controller 创建运行 Heapster Pod：

```
$ kubectl get pod --selector k8s-app=heapster --namespace=kube-system
NAME                READY     STATUS    RESTARTS   AGE
heapster-v10-s8jc1  1/1       Running   0          44s
```

Heapster 运行后会收集 Kubernetes 的监控数据，导入到 InfluxDB，然后就可以通过 Grafana 查看数据图表展示，如图 2-4 所示。

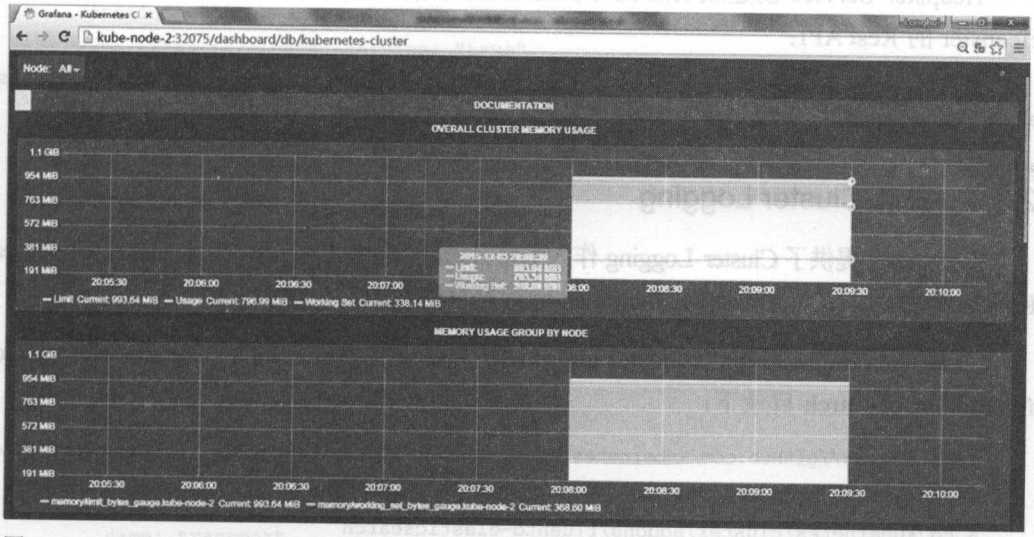


图 2-4 Grafana 展示 Kubernetes 监控

Heapster Service 的定义文件 influxdb/heapster-service.yaml：

```
kind: Service
apiVersion: v1
metadata:
  name: heapster
  namespace: kube-system
```

```
    labels:
      kubernetes.io/cluster-service: "true"
      kubernetes.io/name: "Heapster"
  spec:
    ports:
      - port: 80
        targetPort: 8082
    selector:
      k8s-app: heapster
```

通过定义文件创建 Heapster Service:

```
$ kubectl create -f influxdb/heapster-service.yaml
service "heapster" created
```

Heapster Service 创建成功后就可以通过 Kubernetes API Server 提供的 Proxy 接口调用 Heapster 的 Rest API:

```
$ curl http://kube-master:8080/api/v1/proxy/namespaces/kube-system/services/heapster/
api/v1/...
```

2.3.3 安装 Cluster Logging

Kubernetes 提供了 Cluster Logging 作为平台日志，Cluster Logging 使用 Fluentd+Elasticsearch+Kibana 来收集、汇总和展示 Kubernetes 运行平台的日志。

下载 Kubernetes 发布包，Cluster Logging 扩展插件在发布包的 Kubernetes/cluster/addons/fluentd-elasticsearch 目录下:

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/fluentd-elasticsearch
```

部署 Elasticsearch

Elasticsearch Replication Controller 的定义文件 es-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: elasticsearch-logging-v1
```

```
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 2
  selector:
    k8s-app: elasticsearch-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: elasticsearch-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - image: gcr.io/google_containers/elasticsearch:1.7
        name: elasticsearch-logging
        resources:
          limits:
            cpu: 100m
        ports:
        - containerPort: 9200
          name: db
          protocol: TCP
        - containerPort: 9300
          name: transport
          protocol: TCP
        volumeMounts:
        - name: es-persistent-storage
          mountPath: /data
      volumes:
      - name: es-persistent-storage
        emptyDir: {}
```

通过定义文件创建 Elasticsearch Replication Controller：

```
$ kubectl create -f es-controller.yaml
replicationcontroller "elasticsearch-logging-v1" created
```

Elasticsearch Replication Controller 创建运行 Elasticsearch Pod：

```
$ kubectl get pod --selector k8s-app=elasticsearch-logging --namespace=kube-system
NAME                            READY     STATUS    RESTARTS   AGE
elasticsearch-logging-v1-joxgq  1/1       Running   0          48s
elasticsearch-logging-v1-ofmn2  1/1       Running   0          48s
```

Elasticsearch Service 的定义文件 es-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-logging
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Elasticsearch"
spec:
  ports:
  - port: 9200
    protocol: TCP
    targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

通过定义文件创建 Elasticsearch Service：

```
$ kubectl create -f es-service.yaml
service "elasticsearch-logging" created
```

Elasticsearch Service 创建成功后，可以通过 Kubernetes API Server 提供的 Proxy 接口访问 Elasticsearch，如图 2-5 所示。

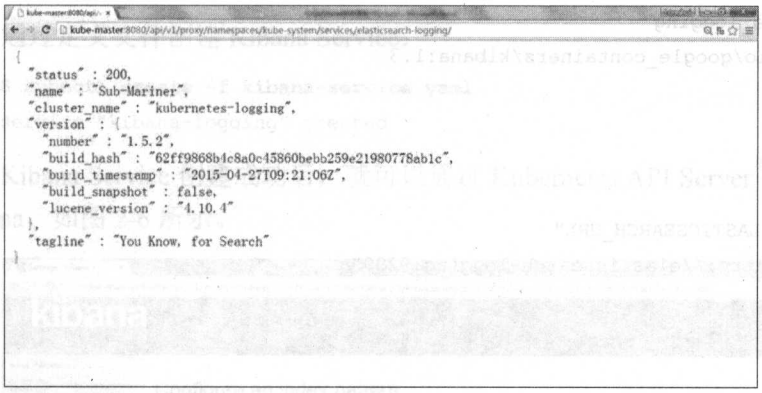


图 2-5 访问 Elasticsearch API

部署 Kibana

Kibana Replication Controller 的定义文件 kibana-controller.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kibana-logging-v1
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kibana-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: kibana-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
```

```
    - name: kibana-logging
      image: gcr.io/google_containers/kibana:1.3
      resources:
        limits:
          cpu: 100m
      env:
        - name: "ELASTICSEARCH_URL"
          value: "http://elasticsearch-logging:9200"
      ports:
      - containerPort: 5601
        name: ui
        protocol: TCP
```

通过定义文件创建 Kibana Replication Controller：

```
$ kubectl create -f kibana-controller.yaml
replicationcontroller "kibana-logging-v1" created
```

Kibana Replication Controller 创建运行 Kibana Pod：

```
$ kubectl get pod --selector k8s-app=kibana-logging --namespace=kube-system
NAME                      READY     STATUS    RESTARTS    AGE
kibana-logging-v1-xqjhz   1/1       Running   1           2m
```

Kibana Service 的定义文件 kibana-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: kibana-logging
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Kibana"
spec:
  ports:
  - port: 5601
    protocol: TCP
    targetPort: ui
  selector:
    k8s-app: kibana-logging
```

通过定义文件创建 Kibana Service：

```
$ kubectl create -f kibana-service.yaml
service "kibana-logging" created
```

Kibana Service 创建成功后，就可以通过 Kubernetes API Server 提供的 Proxy 接口访问 Kibana，如图 2-6 所示。

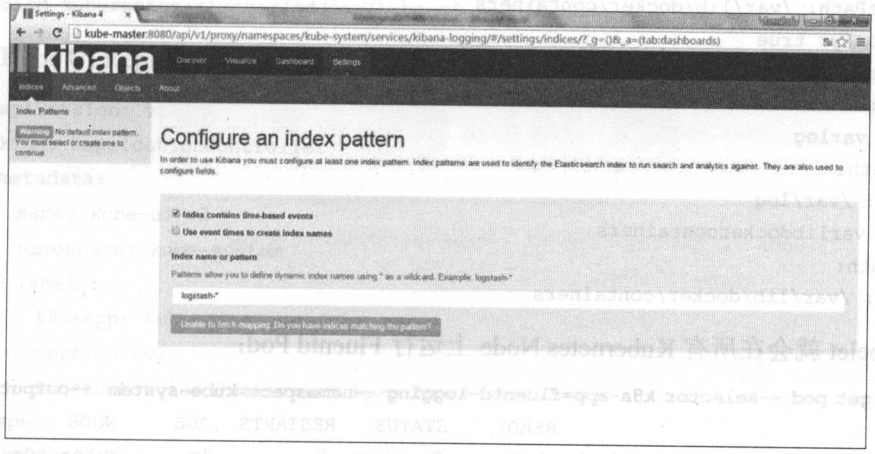


图 2-6 访问 Kibana

部署 Fluentd

Fluentd 作为 Logging Agent 需要运行在所有 Kubernetes 节点上，我们通过 Kubelet 将其作为 Static Pod（参考 12.1.1 节）运行。

在所有 Kubernetes Node 上，在 Kubelet 的目录/etc/kubernetes/manifests 下放入 Fluentd Pod 的定义文件 fluentd-es.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
spec:
  containers:
  - name: fluentd-elasticsearch
    image: gcr.io/google_containers/fluentd-elasticsearch:1.11
    resources:
      limits:
```

```
        cpu: 100m
    args:
    - -q
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

然后 Kubelet 就会在所有 Kubernetes Node 上运行 Fluentd Pod：

```
$ kubectl get pod --selector k8s-app=fluentd-logging --namespace=kube-system --output wide
NAME                                  READY     STATUS    RESTARTS   AGE       NODE
fluentd-elasticsearch-kube-node-1     1/1       Running   0          1m        kube-node-1
fluentd-elasticsearch-kube-node-2     1/1       Running   0          1m        kube-node-2
fluentd-elasticsearch-kube-node-3     1/1       Running   0          1m        kube-node-3
```

Fluentd 运行后，将会收集日志并导入到 Elasticsearch，从而可以通过 Kibana 查询到日志，如图 2-7 所示。

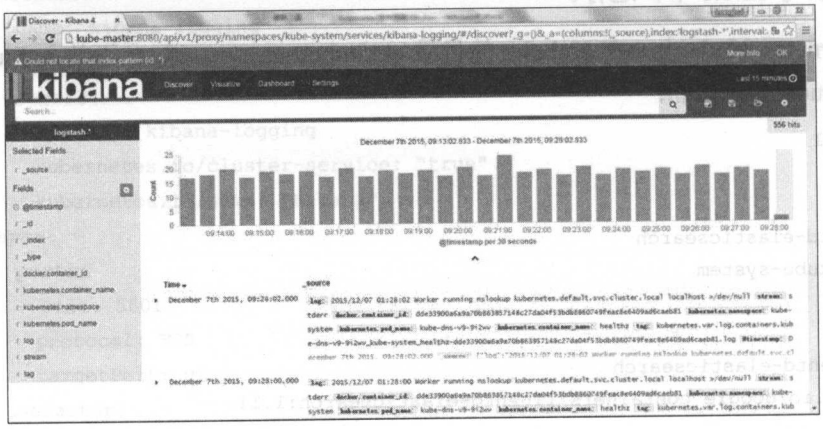


图 2-7　Kibana 查询日志

2.3.4 安装 Kube UI

Kube UI 是 Kubernetes 提供的 Web 管理界面，下载 Kubernetes 发布包，Kube UI 扩展插件在 Kubernetes 发布包的 cluster/addons/kube-ui 目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/kube-ui
```

Kube UI Replication Controller 的定义文件 kube-ui-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-ui-v2
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    version: v2
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-ui
    version: v2
  template:
    metadata:
      labels:
        k8s-app: kube-ui
        version: v2
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: kube-ui
        image: gcr.io/google_containers/kube-ui:v2
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
        ports:
```

```
      - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 30
        timeoutSeconds: 5
```

通过定义文件创建 Kube UI Replication Controller：

```
$ kubectl create -f kube-ui-rc.yaml
replicationcontroller "kube-ui-v2" created
```

Kube UI Replication Controller 创建运行 Kube UI Pod：

```
$ kubectl get pods -l k8s-app=kube-ui --namespace=kube-system
NAME              READY     STATUS       RESTARTS    AGE
kube-ui-v2-6g0og  1/1       Running      0           28s
```

Kube UI Service 的定义文件 kube-ui-svc.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-ui
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeUI"
spec:
  selector:
    k8s-app: kube-ui
  ports:
  - port: 80
    targetPort: 8080
```

通过定义文件创建 Kube UI Service：

```
$ kubectl create -f kube-ui-rc.yaml
replicationcontroller "kube-ui-v2" created
```

Kube UI Service 创建成功后就可以通过 Kubernetes API Server 的接口访问到 Kube UI，如图 2-8 所示。

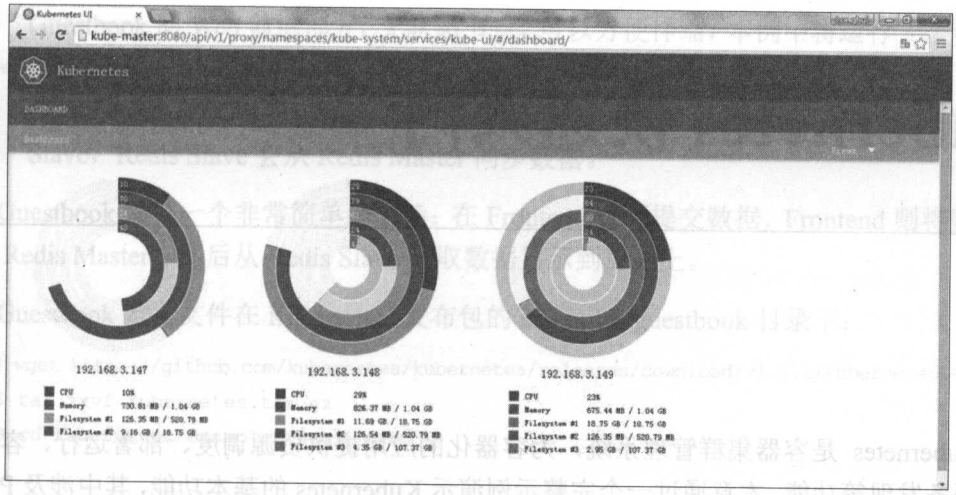


图 2-8　访问 Kube UI

第 3 章

Kubernetes 快速入门

Kubernetes 是容器集群管理系统，为容器化的应用提供资源调度、部署运行、容灾容错和服务发现等功能。本章通过一个完整示例演示 Kubernetes 的基本功能，其中涉及 Pod、Replication Controller 和 Service 这三个 Kubernetes 基本要素的功能展示，从而帮助读者快速理解 Kubernetes。

3.1 示例应用 Guestbook

本章要演示的示例应用是一个名叫 Guestbook 的应用，Guestbook 是一个典型的 Web 应用。Guestbook 的部署运行结构如图 3-1 所示。

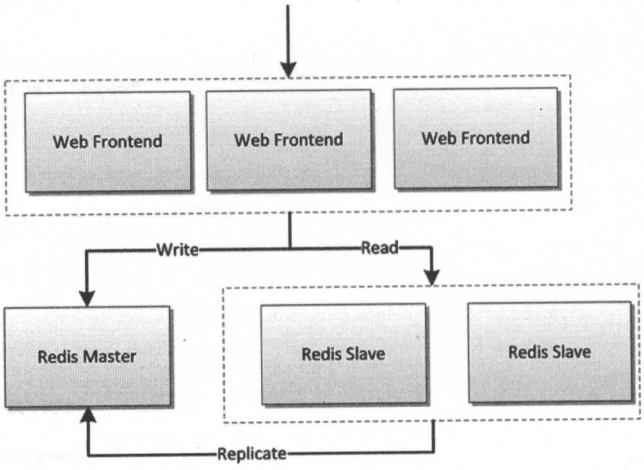


图 3-1 Guestbook 结构

Guestbook 包含两部分。

- Frontend
  Guestbook 的 Web 前端部分，无状态节点，可以方便伸缩，本例中将运行 3 个实例。
- Redis
  Guestbook 的存储部分，Redis 采用主备模式，即运行 1 个 Redis Master 和 2 个 Redis Slave，Redis Slave 会从 Redis Master 同步数据。

Guestbook 提供一个非常简单的功能：在 Frontend 页面提交数据，Frontend 则将数据保存到 Redis Master，然后从 Redis Slave 读取数据显示到页面上。

Guestbook 定义文件在 Kubernetes 发布包的 examples/guestbook 目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/examples/guestbook
```

3.2 准备工作

需要准备一套 Kubernetes 运行环境，可参考 2.2 节的介绍。另外，Kubernetes 需要安装 Cluster DNS，可参考 2.3.1 节的介绍。

本文使用的 Kubernetes 环境信息如下所示。

```
$ kubectl cluster-info
Kubernetes master is running at http://k8s-master:8080
KubeDNS is running at http://k8s-master:8080/api/v1/proxy/namespaces/kube-system/
services/kube-dns
...

$ kubectl -s http://kube-master:8080 get componentstatuses
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok                   nil
scheduler            Healthy   ok                   nil
etcd-0               Healthy   {"health": "true"}   nil

$ kubectl -s http://kube-master:8080 get nodes
NAME          LABELS                                   STATUS    AGE
kube-node-1   kubernetes.io/hostname=kube-node-1   Ready     19m
```

```
kube-node-2   kubernetes.io/hostname=kube-node-2   Ready     18m
kube-node-3   kubernetes.io/hostname=kube-node-3   Ready     18m
```

3.3 运行 Redis

首先在 Kubernetes 上部署运行 Redis，包括 Redis Master 和 Redis Slave。

3.3.1 创建 Redis Master Pod

Pod 是 Kubernetes 的基本处理单元，Pod 包含一个或者多个相关的容器，应用将以 Pod 的形式运行在 Kubernetes 中（本质上是运行容器）。而 Replication Controller 能够控制 Pod 按照指定副本数目持续运行，一般情况下是通过 Replication Controller 来创建 Pod，来保证 Pod 的可靠性。

Redis Master Replication Controller 的定义文件 redis-master-controller.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
      - name: master
        image: redis
        ports:
        - containerPort: 6379
```

在 Kubernetes 中，主要通过文件来定义 API 对象，定义文件形式更加清晰，也方便保

存和修改，定义文件格式支持 JSON 和 YAML，本书主要使用 YAML 格式。

定义文件中首先声明了 API 对象的基本属性：API 版本（.apiVersion）、API 对象类型（.kind）和元数据（.metadata），定义文件 redis-master-controller.yaml 中定义了 V1 版本下一个名称为 redis-master 的 Replication Controller。另外配置了 Replication Controller 的规格（.spec），其中设置了 Pod 的副本数（.spec.replicas）和 Pod 模板（.spec.template）。Pod 模板中说明了 Pod 包含一个容器，该容器使用镜像 redis，即运行 Redis Master，该 Replication Controller 将关联 1 个这样的 Pod，而 Replication Controller 和 Pod 的关联是通过 Label 来实现（.spec.selector 和.spec.template.metadata.labels）的。

通过定义文件创建 Redis Master Replication Controller：

```
$ kubectl create -f redis-master-controller.yaml
replicationcontroller "redis-master" created
```

创建成功后，可查询 Redis Master Replication Controller：

```
$ kubectl get replicationcontroller redis-master
CONTROLLER     CONTAINER(S)   IMAGE(S)   SELECTOR              REPLICAS   AGE
redis-master   master         redis      name=redis-master     1          15s
```

Redis Master Replication Controller 将会创建 1 个 Redis Master Pod，创建出来的 Pod 就会带上 Label name=redis-master：

```
$ kubectl get pod --selector name=redis-master
NAME                 READY     STATUS    RESTARTS   AGE
redis-master-vdkfp   1/1       Running   0          31s
```

Replication Controller 在创建出 Pod 以后，将会保证 Pod 按照指定副本数目持续运行，而通过 Replication Controller 也可以对 Pod 进行一系列操作，包括滚动升级和弹性伸缩等。

3.3.2 创建 Redis Master Service

Kubernetes 中 Pod 是变化的，特别是当受到 Replication Controller 控制的时候，而当 Pod 发生变化的时候，Pod 的 IP 也是变化的。这就导致了一个问题：在 Kubernetes 集群中，Pod 之间如何互相发现并访问呢？比如我们已经运行了 Redis Master Pod，那么 Redis Slave Pod 如何获取 Redis Master Pod 的访问地址呢？为此 Kubernetes 提供了 Service 来实现服务发现。

Kubernetes 中 Service 是真实应用的抽象，将用来代理 Pod，对外提供固定 IP 作为访问入口，这样通过访问 Service 便能访问到相应的 Pod，而对访问者来说只需知道 Service 的访问地址，而不需要感知 Pod 的变化。

上一步中已经运行起 Redis Master Pod，现在创建 Redis Master Service 来代理 Redis Master Pod，Redis Master Service 的定义文件 redis-master-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  ports:
    # the port that this service should serve on
  - port: 6379
    targetPort: 6379
  selector:
    name: redis-master
```

Service 是通过 Label 来关联 Pod 的，在 Service 的定义中，设置.spec.selector 为 name=redis-master，将关联上 Redis Master Pod。

通过定义文件创建 Redis Master Service：

```
$ kubectl create -f redis-master-service.yaml
service "redis-master" created
```

创建成功后查看 Redis Master Service：

```
$ kubectl get service redis-master
NAME           CLUSTER_IP      EXTERNAL_IP   PORT(S)    SELECTOR            AGE
redis-master   10.254.233.212  <none>        6379/TCP   name=redis-master   13s
```

Redis Master Service 的查询信息中显示属性 CLUSTER_IP 为 10.254.233.212，属性 PORT(S)为 6379/TCP，其中 10.254.233.212 是 Kubernetes 分配给 Redis Master Service 的虚拟 IP，6379/TCP 则是 Service 会转发的端口（通过 Service 定义文件中的.spec.ports[0].port 指定），Kubernetes 会将所有访问 10.254.233.212:6379 的 TCP 请求转发到 Redis Master Pod 中，目标端口是 6379/TCP（通过 Service 定义文件中的 spec.ports[0].targetPort 指定）。

因为创建了 Redis Master Service 来代理 Redis Master Pod，所以 Redis Slave Pod 通过 Redis Master Service 的虚拟 IP 10.254.233.212 就可以访问到 Redis Master Pod，但是如果只是硬配置 Service 的虚拟 IP 到 Redis Slave Pod 中，这样还不是真正的服务发现，Kubernetes 提供了两种发现 Service 的方法。

- **环境变量**

当 Pod 运行的时候，Kubernetes 会将之前存在的 Service 的信息通过环境变量写到 Pod 中，以 Redis Master Service 为例，它的信息会被写到 Pod 中：

```
REDIS_MASTER_SERVICE_HOST=10.254.233.212
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.254.233.212:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.254.233.212:6379
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.254.233.212
```

这种方法要求 Pod 必须在 Service 之后启动，之前启动的 Pod 没有这些环境变量。采用 DNS 方式就没有这个限制。

- **DNS**

当有新的 Service 创建时，就会自动生成一条 DNS 记录，以 Redis Master Service 为例，有一条 DNS 记录：

```
redis-master => 10.254.233.212
```

使用这种方法，Kubernetes 需要安装 Cluster DNS，可参考 2.3.1 节的介绍。

3.3.3 创建 Redis Slave Pod

通过 Replication Controller 可创建 Redis Slave Pod，将创建两个 Redis Slave Pod。Redis Slave Replication Controller 的定义文件 redis-slave-controller.yaml：

```
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
```

```
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
    spec:
      containers:
      - name: worker
        image: gcr.io/google_samples/gb-redisslave:v1
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access an environment variable to find the master
          # service's host, comment out the 'value: dns' line above, and
          # uncomment the line below.
          # value: env
        ports:
        - containerPort: 6379
```

Redis Slave Replication Controller 定义中设置 Pod 副本数为 2，而 Pod 模板中包含一个容器，容器使用镜像 gcr.io/google_samples/gb-redisslave:v1，该镜像实际上是基于镜像 redis 重写了启动脚本，将作为 Redis Master 的备节点启动，启动脚本如下：

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
  redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
else
  redis-server --slaveof redis-master 6379
fi
```

其中通过环境变量 GET_HOSTS_FROM 来控制使用环境变量或者 DNS 方式来发现 Redis Master Server。

提示

镜像 gcr.io/google_samples/gb-redisslave:v1 的 Dockerfile 参考：

https://github.com/kubernetes/kubernetes/tree/v1.1.1/examples/guestbook/redis-slave

通过定义文件创建 Redis Slave Replication Controller：

```
$ kubectl create -f redis-slave-controller.yaml
replicationcontroller "redis-slave" created
```

创建成功后，查询 Redis Slave Replication Controller：

```
$ kubectl get replicationcontroller redis-slave
CONTROLLER    CONTAINER(S)   IMAGE(S)                              SELECTOR            REPLICAS   AGE
redis-slave   worker         gcr.io/google_samples/gb-redisslave:v1   name=redis-slave 2          4s
```

Redis Slave Replication Controller 创建运行两个 Redis Slave Pod:

```
$ kubectl get pod --selector name=redis-slave
NAME                READY     STATUS    RESTARTS   AGE
redis-slave-7g617   1/1       Running   0          24s
redis-slave-8dhc2   1/1       Running   0          24s
```

3.3.4 创建 Redis Slave Service

创建 Redis Salve Service 来代理 Redis Salve Pod，Redis Salve Service 的定义文件 redis-slave-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  ports:
    # the port that this service should serve on
  - port: 6379
  selector:
    name: redis-slave
```

通过定义文件创建 Redis Salve Service：

```
$ kubectl create -f redis-slave-service.yaml
service "redis-slave" created
```

查询 Redis Salve Service：

```
$ kubectl get service redis-slave
NAME          CLUSTER_IP       EXTERNAL_IP   PORT(S)    SELECTOR          AGE
redis-slave   10.254.108.203   <none>        6379/TCP   name=redis-slave   4s
```

3.4 运行 Frontend

3.4.1 创建 Frontend Pod

通过 Frontend Replication Controller 来创建 Frontend Pod，将创建 3 个 Frontend Pod。

Frontend Replication Controller 的定义文件 frontend-controller.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
```

```
        # line below.
        # value: env
      ports:
      - containerPort: 80
```

Frontend Replication Controller 的定义中设置 Pod 副本数为 3，Pod 模板包含一个容器，容器使用镜像 gcr.io/google_samples/gb-frontend:v3，这是一个 PHP 实现的 Web 应用，简单地说将写数据到 Redis Master，并从 Redis Slave 中读取数据：

```
<?

set_include_path('.:/usr/local/lib/php');

error_reporting(E_ALL);
ini_set('display_errors', 1);

require 'Predis/Autoloader.php';

Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
  $host = 'redis-master';
  if (getenv('GET_HOSTS_FROM') == 'env') {
    $host = getenv('REDIS_MASTER_SERVICE_HOST');
  }
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
  } else {
    $host = 'redis-slave';
    if (getenv('GET_HOSTS_FROM') == 'env') {
      $host = getenv('REDIS_SLAVE_SERVICE_HOST');
    }
```

```
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
  }
} else {
  phpinfo();
} ?>
```

代码中是通过环境变量 GET_HOSTS_FROM 来控制使用环境变量方式还是 DNS 方式来发现 Redis Master Server 和 Redis Slave Server 的。

提示

镜像 gcr.io/google_samples/gb-frontend:v3 的 Dockerfile 参考：
https://github.com/kubernetes/kubernetes/tree/v1.1.1/examples/guestbook/php-redis

通过定义文件创建 Frontend Replication Controller：

```
$ kubectl create -f frontend-controller.yaml
replicationcontroller "frontend" created
```

创建成功后，查询 Frontend Replication Controller：

```
$ kubectl get replicationcontroller frontend
CONTROLLER  CONTAINER(S)  IMAGE(S)                              SELECTOR        REPLICAS  AGE
frontend    php-redis     gcr.io/google_samples/gb-frontend:v3  name=frontend   3         9s
```

Frontend Replication Controller 创建运行 3 个 Frontend Pod：

```
$ kubectl get pod --selector name=frontend
NAME              READY     STATUS    RESTARTS   AGE
frontend-1670s    1/1       Running   0          22s
frontend-2t6sw    1/1       Running   0          21s
frontend-ehbxc    1/1       Running   0          21s
```

3.4.2 创建 Frontend Service

创建 Frontend Service 代理 Frontend Pod，Frontend Service 的定义文件 frontend-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  ports:
    # the port that this service should serve on
  - port: 80
  selector:
    name: frontend
```

通过定义文件创建 Frontend Service：

```
$ kubectl create -f frontend-service.yaml
service "frontend" created
```

查询 Frontend Service：

```
$ kubectl get service frontend
NAME       CLUSTER_IP      EXTERNAL_IP   PORT(S)    SELECTOR          AGE
frontend   10.254.69.78    <none>        80/TCP     name=frontend     22s
```

3.5 设置 Guestbook 外网访问

至此 Guestbook 就已经运行在 Kubernetes 上，但是还有一件很重要的事情要解决，用户该如何访问 Guestbook Frontend 呢？是否通过 Frontend Service 的虚拟 IP 10.254.69.78？但是 Service 的虚拟 IP 是由 Kubernetes 虚拟出来的内部网络，而外部网络是无法寻址到的，这时候就需要增加一层网络转发，即外网到内网的转发。实现方式有很多种，我们这里采用一种叫作 NodePort 的方式来实现。即 Kubernetes 将会在每个 Node 上设置端口，称为 NodePort，通过 NodePort 端口可以访问到 Pod。

修改 Frontend Service 的定义文件 frontend-service.yaml，设置 spec.type 为 NodePort：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  type: NodePort
  ports:
    # the port that this service should serve on
    - port: 80
  selector:
    name: frontend
```

重新创建 Frontend Service：

```
$ kubectl replace -f frontend-service.yaml --force
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:31505) to serve traffic.

See http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md for more
details.
service "frontend" created
```

Frontend Service 创建成功信息中说明了已经创建 NodePort (tcp:31505)，那么通过任何一个 Node 的 IP 和 NodePort (tcp:31505)即可访问到 Guestbook Frontend。然后在界面的输入框中输入任意字符串并提交，字符串将会被保存并显示在最下方，如图 3-2 所示。

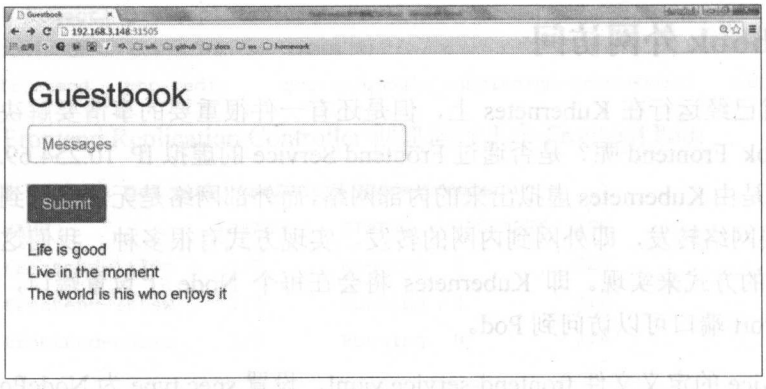


图 3-2　Guestbook 界面

3.6 清理 Guestbook

清理 Guestbook，只需要分别删除创建出的 Replication Controller 和 Service：

```
$ kubectl delete replicationcontroller redis-master redis-slave frontend
replicationcontroller "redis-master" deleted
replicationcontroller "redis-slave" deleted
replicationcontroller "frontend" deleted

$ kubectl delete service redis-master redis-slave  frontend
service "redis-master" deleted
service "redis-slave" deleted
service "frontend" deleted
```

第4章

Pod

Pod 是 Kubernetes 的基本操作单元，也是应用运行的载体。整个 Kubernetes 系统都是围绕着 Pod 展开的，比如如何部署运行 Pod、如何保证 Pod 的可靠性、如何访问 Pod 等。另外，Pod 是一个或多个相关容器的集合，这可以说是一大创新点，提供了一种容器组合的模型，当然也使得在 Pod 的操作和生命周期管理上稍有不同。本章将围绕 Pod 进行详细讲解，首先介绍如何运行一个最基本的 Pod，然后由浅入深地说明 Pod 的各个方面，包括资源隔离、网络、生命周期管理和调度。

4.1 国际惯例的 Hello World

不免俗地我们也将以 Hello World 作为开始，创建一个简单的 Hello World Pod，运行一个输出“Hello World”的容器，Hello World Pod 的定义文件 hello-world-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: OnFailure
  containers:
  - name: hello
    image: "ubuntu:14.04"
```

```
  command: ["/bin/echo","Hello","World"]
```

定义文件中描述了 Pod 的属性和行为，其中的主要要素如下所示。

- apiVersion：声明 Kubernetes 的 API 版本，目前是 v1。
- kind：声明 API 对象的类型，这里类型是 Pod。
- metadata：设置 Pod 的元数据。
  - name：指定 Pod 的名称，Pod 名称必须在 Namespace 内唯一。
- spec：配置 Pod 的具体规格。
  - restartPolicy：设置 Pod 的重启策略。
  - containers：设置 Pod 中容器的规格，数组形式，每一项定义一个容器。
    name：指定容器的名称，在 Pod 的定义中唯一。
    image：设置容器镜像。
    command：设置容器的启动命令

Hello World Pod 的定义中设置了一个容器，名称是 hello，使用镜像 ubuntu:14.04，同时启动命令是["/bin/echo","Hello","World"]，实际上这类似我们使用 docker run 命令运行容器：

```
$ docker run --name hellop  ubuntu:14.04 /bin/echo Hello World
Hello World
```

需要注意的是，因为容器输出完 Hello World 就会退出，这是一次性执行的，所以在 Pod 的定义中，.spec.restartPolicy 设置为 OnFailure，即在容器正常退出的情况下不会重新创建容器。

通过定义文件创建 Hello World Pod：

```
$ kubectl create -f hello-world-pod.yaml
pod "hello-world" created
```

创建成功后，可以查询 Hello World Pod：

```
$ kubectl get pod hello-world
NAME          READY     STATUS     RESTARTS   AGE
hello-world   0/1       ExitCode:0   0          1m
```

然后可以查询 Pod 输出：

```
$ kubectl logs hello-world
Hello World
```

最后删除 Hello World Pod：

```
$ kubectl delete pod hello-world
pod "hello-world" deleted
```

4.2 Pod 的基本操作

4.2.1 创建 Pod

4.1 节中我们使用 kubectl create 命令创建了 Pod，Kubernetes 中大部分 API 对象都是通过 kubectl create 命令创建的。

如果 Pod 的定义存在错误，kubectl create 会打印错误信息，现有一个 Pod 的错误定义文件 error-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
spec:
  restartPolicy: Maybe
  containers:
  - name: hello
    image: "ubuntu:14.04"
    entrypoint : ["/bin/echo","Hello","World"]
```

通过定义文件创建 Pod，将会创建失败并提示相应的错误信息：

```
$ kubectl create -f error-pod.yaml
The Pod "" is invalid.

* metadata.name: required value, Details: name or generateName is required
* spec.restartPolicy: unsupported value 'Maybe', Details: supported values: Always,
OnFailure, Never
```

4.2.2 查询 Pod

最常用的查询命令就是 kubectl get，可以查询一个或者多个 Pod 的信息，现在查询指定 Pod：

```
$ kubectl get pod my-pod
NAME      READY     STATUS    RESTARTS   AGE
my-pod    1/1       Running   0          10s
```

查询显示的字段含义如下所示。

- NAME：Pod 的名称。
- READY：Pod 的准备状况，右边的数字表示 Pod 包含的容器总数目，左边的数字表示准备就绪的容器数目。
- STATUS：Pod 的状态。
- RESTARTS：Pod 的重启次数。
- AGE：Pod 的运行时间。

其中 Pod 的准备状况指的是 Pod 是否准备就绪以接收请求，Pod 的准备状况取决于容器，即所有容器都准备就绪了，Pod 才准备就绪。这时候 Kubernetes 的代理服务才会添加 Pod 作为分发后端，而一旦 Pod 的准备状况变为 false（至少一个容器的准备状况变为 false），Kubernetes 会将 Pod 从代理服务的分发后端移除，即不会分发请求给该 Pod。

默认情况下，kubectl get 只是显示 Pod 的简要信息，以下方式可用于获取 Pod 的完整信息：

```
$ kubectl get pod my-pod --output json  #用 JSON 格式显示 Pod 的完整信息
$ kubectl get pod my-pod --output yaml  #用 YAML 方式显示 Pod 的完整信息
```

另外，kubectl get 支持以 Go Template 方式过滤出指定的信息，比如查询 Pod 的运行状态：

```
$ kubectl get pods my-pod --output=go-template --template={{.status.phase}}
Running
```

另一个命令 kubectl describe 支持查询 Pod 的状态和生命周期事件：

```
$ kubectl describe pod my-pod
Name:              my-pod
Namespace:         default
Image(s):          nginx
Node:              kube-node-2/192.168.3.148
Start Time:        Sun, 22 Nov 2015 18:00:34 +0800
Labels:            <none>
Status:            Running
```

```
Reason:
Message:
IP:             10.0.62.37
Replication Controllers:    <none>
Containers:
  container1:
    Container ID:
docker://60387df7e5f2c781ed508084ffa3579f05482c6b465abb3d4fcae0ade064b813
    Image:      nginx
    Image ID:
docker://6886fb5a9b8d73b12d842bab8f9a6941c36094c2974abddb685d54c9d99e37da
    State:      Running
      Started:  Sun, 22 Nov 2015 18:00:42 +0800
    Ready:      True
    Restart Count:  0
    Environment Variables:
Conditions:
  Type    Status
  Ready   True
Volumes:
  ...
Events:
  ...
```

查询显示的字段含义如下所示。

- Name：Pod 的名称。
- Namespace：Pod 的 Namespace。
- Image(s)：Pod 使用的镜像。
- Node：Pod 所在的 Node。
- Start Time：Pod 的起始时间。
- Labels：Pod 的 Label。
- Status：Pod 的状态。
- Reason：Pod 处于当前状态的原因。
- Message：Pod 处于当前状态的信息。
- IP：Pod 的 PodIP。
- Replication Controllers：Pod 对应的 Replication Controller。

- Containers：Pod 中容器的信息。
  - Container ID：容器的 ID。
  - Image：容器的镜像。
  - Image ID：镜像的 ID。
  - State：容器的状态。
  - Ready：容器的准备状况（true 表示准备就绪）。
  - Restart Count：容器的重启次数统计。
  - Environment Variables：容器的环境变量。
- Conditions：Pod 的条件，包含 Pod 的准备状况（true 表示准备就绪）。
- Volumes：Pod 的数据卷。
- Events：与 Pod 相关的事件列表。

4.2.3 删除 Pod

可以通过 kubectl delete 命令删除 Pod：

```
$ kubectl delete pod my-pod
```

另外，kubectl delete 命令可以批量删除全部 Pod：

```
$ kubectl delete pod --all
```

4.2.4 更新 Pod

Pod 在创建之后如果希望更新 Pod，可以在修改 Pod 的定义文件后执行：

```
$ kubectl replace /path/to/my-pod.yaml
```

但是因为 Pod 的很多属性是没办法修改的，比如容器镜像，这时候可以通过 kubectl replace 命令设置--force 参数，等效于重建 Pod。

4.3 Pod 与容器

在 Docker 中，容器是最小处理单位，增删改查的对象是容器，容器是一种虚拟化技术，容器之间是隔离的，隔离是基于 Linux Namespace 实现的，Linux 内核中提供了 6 种 Linux Namespace 隔离的系统调用，如表 4-1 所示。

表 4-1 Linux Namespace

Linux Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

而在 Kubernetes 中，Pod 包含一个或者多个相关的容器，Pod 可以认为是容器的一种延伸扩展，一个 Pod 也是一个隔离体，而 Pod 包含的一组容器又是共享的（当前共享的 Linux Namespace 包括：PID、Network、IPC 和 UTS）。除此之外，Pod 中的容器可以访问共同的数据卷来实现文件系统的共享，所以 Kubernetes 中的数据卷是 Pod 级别的，而不是容器级别的。

Pod 是容器的集合，容器是真正的执行体。相比原生的容器接口，Pod 提供了更高层次的抽象，但是 Pod 的设计并不是为了运行同一个应用的多个实例，而是运行一个应用多个紧密联系的程序。而每个程序运行在单独的容器中，以 Pod 的形式组合成一个应用。相比于在单个容器中运行多个程序，这样的设计有以下好处。

- 透明性：将 Pod 内的容器向基础设施可见，底层系统就能向容器提供如进程管理和资源监控等服务，这样能给用户带来极大便利。
- 解绑软件的依赖：这样单个容器可以独立地重建和重新部署，可以实现独立容器的实时更新。
- 易用性：用户不需要运行自己的进程管理器，也不需负责信号量和退出码的传递等。
- 高效性：因为底层设备负责更多的管理，容器因而能更轻量化。

在 Pod 中可以详细地配置如何运行一个容器，就像我们使用 docker run 命令运行容器一样，接下来结合实例说明如何在 Pod 中定义容器。

4.3.1 镜像

运行容器必须先指定镜像，镜像的名称则遵循 Docker 的命名规范。运行容器前需要本地存在对应的镜像，如果镜像不存在，会从 Docker 镜像仓库下载。Kubernetes 中可以选择

镜像的下载策略，支持的策略如下。

- Always：每次都下载最新的镜像。
- Never：只使用本地镜像，从不下载。
- IfNotPresent：只有当本地没有的时候才下载镜像。

Kubernetes Node 是容器运行的宿主机，Pod 被分配到 Node 之后，会根据镜像下载策略选择是否下载镜像。有时候网络下载是一个较大的开销，可以根据需要自行选择策略，但是无论如何要确保镜像在本地或者镜像仓库存在，否则 Pod 无法运行。

Pod 定义中一个容器镜像的配置示例如下所示：

```
name: hello
image: "ubuntu:14.04"
imagePullPolicy: Always
```

Docker 镜像仓库也称为 Docker 注册服务器（Docker Registry），它包括 Docker 公共镜像仓库（Docker Hub）和私有镜像仓库。

例如对于 ubuntu:14.04，它实际上等效于 docker.io/ubuntu:14.04，即从 Docker 公共镜像仓库下载镜像。而对于 myregistry.com/ubuntu:14.04 来说，myregistry.com 是 Docker 私有镜像仓库地址，即从 myregistry.com 下载镜像。

使用 Docker 私有镜像仓库，往往需要进行认证。一种方法是在所有的 Node 上手工操作 docker login [registry]进行登录认证；另一种方法是在 Pod 中添加 Image Pull Secret 用于认证，Image Pull Secret 是一种 kubernetes.io/dockercfg 类型的 Secret，Kubernetes 用来进行 Docker 镜像仓库的认证，具体操作方法如下所示。

1. 首先登录 Docker 私有注册服务器，例如 myregistry.com：

```
$ docker login myregistry.com
WARNING: The Auth config file is empty
Username: admin
Password:
Email: wlh6666@qq.com
WARNING: login credentials saved in /root/.dockercfg.
Login Succeeded
```

登录成功后，Docker 会生成一个文件.dockercfg：

```
$ echo $(cat ~/.dockercfg)
```

```
{ "myregistry.com": { "auth": "d2FlYWRtaW46d2FlQDEyMw==", "email": "wlh6666@qq.com" } }
```

2. 使用 Base64 编码.dockercfg 内容：

```
$ export DOCKER_CFG_DATA=`cat ~/.dockercfg | base64`
```

根据编码后的内容创建 Image Pull Secret 的定义文件：

```
$ cat > ./image-pull-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockercfg: ${DOCKER_CFG_DATA}
type: kubernetes.io/dockercfg
EOF
```

通过定义文件创建 Image Pull Secret：

```
$ kubectl create -f ./image-pull-secret.yaml
secret "myregistrykey" created
```

然后在 Pod 的定义中通过.spec.imagePullSecrets 添加 Image Pull Secret：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: myregistrykey.com/redis
  restartPolicy: Always
  imagePullSecrets:
    - name: myregistrykey
```

提示

系统管理员可以设置全局的 Image Pull Secret，创建 Pod 的时候会自动添加上 Image Pull Secret。这样一来就不需要每个 Pod 显示配置，可参考 10.3.3 节。

4.3.2 启动命令

启动命令用来说明容器是如何运行的，在 Pod 的定义中可以设置容器启动命令和参数，4.1 节中的 Hello World Pod 就对容器设置了启动命令：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: "ubuntu:14.04"
    command: ["/bin/echo","Hello","World"]
```

另外，容器的启动命令也可以配置为:

```
    command: ["/bin/echo"]
    args: ["Hello","World"]
```

在使用 docker run 命令运行容器的时候，如果没有指定容器的启动命令，容器则使用 Docker 镜像默认的启动命令。这一般是通过 Dockerfile 中的 CMD 和 ENTRYPOINT 进行设置的，这是非常容易混淆的两个概念，假设镜像 image1 的 Dockerfile 声明 CMD 命令如下：

```
ROM ubuntu
CMD ["echo"]
```

运行容器：

```
$ docker run image1 echo hello
hello
```

另一个镜像 image2 的 Dockerfile 声明 ENTRYPOINT 命令如下：

```
FROM ubuntu
ENTRYPOINT ["echo"]
```

运行容器：

```
$ docker run image2 echo hello
echo hello
```

从示例中可以看出，CMD 命令是可覆盖的，docker run 指定的启动命令会把 CMD 设

置的命令覆盖。而 ENTRYPOINT 设置的命令只是一个入口，docker run 指定的启动命令作为参数传递给 ENTRYPOINT 设置的命令，而不是进行替换。

在 Pod 的定义中，command 和 args 都是可选项，将和 Docker 镜像的 ENTRYPOINT 和 CMD 相互作用，生成最终容器的启动命令，具体规则如下所示：

- 如果容器没有指定 command 和 args，则容器使用镜像的 ENTRYPOINT 和 CMD 作为启动命令运行。
- 如果容器指定了 command，而没有指定 args，则容器忽略镜像的 ENTRYPOINT 和 CMD，使用指定的 command 作为启动命令运行。
- 如果容器没有指定 command，只是指定了 args，容器将使用镜像的 ENTRYPOINT 和 CMD 作为启动命令运行。
- 如果容器指定了 command 和 args，则容器使用 command 和 args 作为启动命令运行。

表 4-2 枚举了 4 条规则的相应示例。

表 4-2　容器命令示例

镜像ENTRYPOINT	镜像CMD	容器command	容器args	最终启动命令
[/ep-1]	[foo bar]	NULL	NULL	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	NULL	[ep-2]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-2 zoo boo]

4.3.3　环境变量

Pod 定义中可以设置容器运行时的环境变量：

```
env:
- name: PARAMETER_1
  value: value_1
- name: PARAMETER_2
  value: value_2
```

对于 Hello World Pod，可以通过设置环境变量的方式进行改造：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: hello-world
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: "ubuntu:14.04"
    env:
    - name: MESSAGE
      value: "hello world"
    command: ["/bin/sh", "-c"]
    args: ["/bin/echo \"${MESSAGE}\""]
```

在一些场景下，Pod中的容器希望获取本身的信息，比如Pod的名称、Pod所在的Namespace等。在Kubernetes中提供了Downward API获取这些信息，并且可以通过环境变量告诉容器目前支持的信息。

- Pod的名称：metadata.name。
- Pod的Namespace：metadata.namespace。
- Pod的PodIP：status.podIP。

现在创建一个Pod并通过环境变量来获取Downward API，Pod的定义文件downwardapi-env.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-env
spec:
  containers:
    - name: test-container
      image: ubuntu:14.04
      command: ["/bin/bash","-c","while true; do sleep 5; done"]
      env:
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
```

```
        fieldRef:
          fieldPath: metadata.namespace
      - name: MY_POD_IP
        valueFrom:
          fieldRef:
            fieldPath: status.podIP
```

Pod 创建运行后，查询 Pod 的输出，过滤出配置的 3 个环境变量：

```
$ kubectl exec downwardapi-env env|grep MY_POD
MY_POD_NAME=downwardapi-env
MY_POD_NAMESPACE=default
MY_POD_IP=10.0.10.103
```

4.3.4 端口

在使用 docker run 运行容器的时候往往通过--publish/-p 参数设置端口映射规则，同样的，可以在 Pod 的定义中设置容器的端口映射规则，比如下面这个 Pod 的设置容器 nginx 的端口映射规则为 0.0.0.0:80->80/TCP：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
      hostIP: 0.0.0.0
      hostPort: 80
```

在 Pod 的定义中，通过.spec.containers[].ports[]设置容器的端口，数组形式，每一项的参数如下所示。

- name：设置端口名称，必须在 Pod 内唯一，当只配置一个端口的时候，这是一个可选项，当配置多个端口的时候，这是一个必选项。

- containerPort：必选项，设置在容器内的端口，有效值范围为0~65536（不包括0和65536）。
- protocol：可选项，设置端口的协议，TCP或者UDP，默认是TCP。
- hostIP：可选项，设置在宿主机上的IP，默认绑定到所有可用的IP接口上，即0.0.0.0。
- hostPort：可选项，设置在宿主机上的端口，如果设置则进行端口映射，有效值范围为0~ 65536（不包括0和65536）。

使用宿主机端口需要考虑端口冲突问题，幸运的是，Kubernetes在调度Pod的时候，会检查宿主机端口是否冲突。比如两个Pod都需要使用宿主机端口80，那么调度的时候就会将这两个Pod调度到不同Node上。不过，如果所有Node的端口都被占用了，那么Pod调度会失败。

4.3.5 数据持久化和共享

容器是临时存在的，如果容器被销毁，容器中的数据将会丢失。为了能够持久化数据以及共享容器间的数据，Docker提出了数据卷（Volume）的概念。简单来说，数据卷就是目录或者文件，它可以绕过默认的联合文件系统，而以正常的文件或者目录的形式存在于宿主机上。

在使用docker run运行容器的时候，我们经常使用参数--volume/-v创建数据卷，即将宿主机上的目录或者文件挂载到容器中。即使容器被销毁，数据卷中的数据仍然保存在宿主机上。

一方面，在Kubernetes中对Docker数据卷进行了扩展，支持对接第三方存储系统。另一方面，Kubernetes中的数据卷是Pod级别的，Pod中的容器可以访问共同的数据卷，实现容器间的数据共享。

我们再次对Hello World Pod进行改造。在Pod中声明创建数据卷，Pod中的两个容器将共享数据卷，容器write写入数据，容器hello读出数据，Hello World Pod的定义文件hello-world-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
```

```
  restartPolicy: Never
  containers:
  - name: write
    image: "ubuntu:14.04"
    command: ["bash","-c","echo \"Hello World\" >> /data/hello"]
    volumeMounts:
      - name: data
        mountPath: /data
  - name: hello
    image: "ubuntu:14.04"
    command: ["bash", "-c","sleep 10; cat /data/hello"]
    volumeMounts:
      - name: data
        mountPath: /data
  volumes:
  - name: data
    hostPath:
     path: /tmp
```

可以看到在 Pod 定义中，.spec.volumes 配置了一个名称为 data 的数据卷，数据卷的类型是 hostPath，使用宿主机的目录/tmp。Pod 中的两个容器都通过.spec.containers[].volumeMounts 来设置挂载数据卷到容器中的路径/data。容器 write 将往/data/hello 写入“Hello World”，容器 hello 等待一会儿，然后读取文件/data/hello 的数据显示，即输出“Hello World”。这样一来就实现了两个容器的数据共享。Kubernetes 数据卷提供了非常丰富的持久化支持，详情可参考 7.3 节。

4.4 Pod 的网络

Pod 中的所有容器网络都是共享的，一个 Pod 中的所有容器中的网络是一致的，它们能够通过本地地址（localhost）访问其他用户容器的端口。

在 Kubernetes 网络模型中，每一个 Pod 都拥有一个扁平化共享网络命名空间的 IP，称为 PodIP。通过 PodIP，Pod 就能够跨网络与其他物理机和容器进行通信。

在 Pod 运行后，我们可以查询 Pod 的 PodIP：

```
$ kubectl get pod my-app --template={{.status.podIP}}
10.0.10.204
```

这是一个 10.0.10.0/24 网段的 IP，实际上这是由 Docker 为容器进行网络虚拟化隔离而分配的内部 IP。也可以设置 Pod 为 Host 网络模式，即直接使用宿主机的网络，不进行网络虚拟化隔离。这样一来，Pod 中的所有容器就直接暴露在宿主机的网络环境中，这时候，Pod 的 PodIP 就是其所在 Node 的 IP。

下面定义的 Pod 设置为 Host 网络模式（.spec.hostNetwork=true）：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: app
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: tcp
  hostNetwork: true
```

使用 Host 网络模式需要特别注意，一方面，因为不存在网络隔离，容易发生端口冲突；另一方面，Pod 可以直接访问宿主机上的所有网络设备和服务，从安全性上来说这是不可控的。

4.5 Pod 的重启策略

Pod 的重启策略指的是当 Pod 中的容器终止退出后，重启容器的策略。需要注意的是，因为 Docker 容器的轻量级，重启容器的做法实际上是直接重建容器，所以容器中的数据将会丢失，如有需要持久化的数据，那么需要使用数据卷进行持久化设置。

重启策略是通过 Pod 定义中的.spec.restartPolicy 进行设置的，目前支持以下 3 种策略。

- Always：当容器终止退出后，总是重启容器，默认策略。
- OnFailure：当容器终止异常退出（退出码非 0）时，才重启容器。
- Never：当容器终止退出时，从不重启容器。

现在创建一个 Pod，其中的容器将异常退出（exit 1），而 Pod 的重启策略为 OnFailure，

Pod 的定义文件 on-failure-restart-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: on-failure-restart-pod
spec:
  containers:
  - name: container
    image: ubuntu:14.04
    command: ["bash","-c", "exit 1"]
  restartPolicy: OnFailure
```

通过定义文件创建 Pod：

```
$ kubectl create -f on-failure-restart-pod.yaml
pods/on-failure-restart-pod
```

Pod 创建成功后，一段时间后查询 Pod：

```
$ kubectl get pod on-failure-restart-pod
NAME                      READY     STATUS    RESTARTS   AGE
on-failure-restart-pod    0/1       Error     3          2m
```

在 Pod 的查询信息中，属性 RESTARTS 的值为 3，说明 Pod 中的容器已经重启，可以分别查询每个容器的重启次数：

```
$ kubectl get pod on-failure-restart-pod \
--template="{{range .status.containerStatuses}}{{.name}}:{{.restartCount}}{{end}}"
container:3
```

提示

关于 Pod 中容器的重启次数统计，实际上并不是非常精确，只能作为一个参考。首先它获取 Pod 所在 Node 的所有容器（包括运行和停止的）作为统计基数，所以如果将退出容器进行清理（可能是人工删除，另外 Kubelet 会根据配置定时进行清理），将会减少统计的基数。

4.6 Pod 的状态和生命周期

4.6.1 容器状态

Pod 的本质是一组容器，Pod 的状态便是容器状态的体现和概括，同时容器的状态变化会影响 Pod 的状态变化，触发 Pod 的生命周期阶段转换。

在使用 docker run 运行容器的时候，首先会下载容器镜像。下载成功后运行容器，当容器运行结束退出后（包括正常和异常退出），容器终止，这是一个容器的生命周期过程。相应的，Kubernetes 中对于 Pod 中的容器进行了状态的记录，其中每种状态下包含的信息如下所示。

- Waiting：容器正在等待创建，比如正在下载镜像。
  - Reason：等待的原因。
- Running：容器已经创建，并且正在运行。
  - startedAt：容器创建时间。
- Terminated：容器终止退出。
  - exitCode：退出码。
  - signal：容器退出信号。
  - reason：容器退出原因。
  - message：容器退出信息。
  - startedAt：容器创建时间。
  - finishedAt：容器退出时间。
  - containerID：容器的 ID。

Pod 运行后，可以查询其中容器的状态：

```
$ kubectl describe pod my-pod
...
Containers:
  container1:
    Container ID:
docker://60387df7e5f2c781ed508084ffa3579f05482c6b465abb3d4fcae0ade064b813
    Image:          nginx
    Image ID:
docker://6886fb5a9b8d73b12d842bab8f9a6941c36094c2974abddb685d54c9d99e37da
    State:          Running
```

```
    Started:  Sun,  22 Nov 2015 18:00:42 +0800
    Ready:    True
    Restart Count:  0
    Environment Variables:
...
```

4.6.2 Pod 的生命周期阶段

Pod 的生命周期可以简单描述为：首先 Pod 被创建，紧接着 Pod 被调度到 Node 进行部署运行。Pod 是非常忠诚的，一旦被分配到 Node 后，就不会离开这个 Node，直到它被删除，生命周期完结。

Pod 的生命周期被定义为以下几个阶段。

- Pending：Pod 已经被创建，但是一个或者多个容器还未创建，这包括 Pod 调度阶段，以及容器镜像的下载过程。
- Running：Pod 已经被调度到 Node，所有容器已经创建，并且至少一个容器在运行或者正在重启。
- Succeeded：Pod 中所有容器正常退出。
- Failed：Pod 中所有容器退出，至少有一个容器是一次退出的。

可以查询 Pod 处于生命周期的哪个阶段：

```
$ kubectl get pods my-app --template="{{.status.phase}}"
Running
```

Pod 被创建成功后，首先会进入 Pending 阶段，然后被调度到 Node 后运行，进入 Running 阶段。如果 Pod 中的容器停止（正常或者异常退出），那么 Pod 根据重启策略的不同会进入不同的阶段，举例如下。

- Pod 是 Running 阶段，含有一个容器，容器正常退出：
  如果重启策略是 Always，那么会重启容器，Pod 保持 Running 阶段。
  如果重启策略是 OnFailure，Pod 进入 Succeeded 阶段。
  如果重启策略是 Never，Pod 进入 Succeeded 阶段。
- Pod 是 Running 阶段，含有一个容器，容器异常退出：
  如果重启策略是 Always，那么会重启容器，Pod 保持 Running 阶段。
  如果重启策略是 OnFailure，Pod 保持 Running 阶段。

如果重启策略是 Never，Pod 进入 Failed 阶段。

- Pod 是 Running 阶段，含有两个容器，其中一个容器异常退出：
  如果重启策略是 Always，那么会重启容器，Pod 保持 Running 阶段。
  如果重启策略是 OnFailure，Pod 保持 Running 阶段。
  如果重启策略是 Never，Pod 保持 Running 阶段。
- Pod 是 Running 阶段，含有两个容器，两个容器都异常退出：
  如果重启策略是 Always，那么会重启容器，Pod 保持 Running 阶段。
  如果重启策略是 OnFailure，Pod 保持 Running 阶段。
  如果重启策略是 Never，Pod 进入 Failed 阶段。

一旦被分配到 Node，Pod 就不会离开这个 Node，直到被删除。删除可能是人为地删除，或者被 Replication Controller 删除，也有可能是当 Pod 进入 Succeeded 或者 Failed 阶段过期，被 Kubernetes 清理掉。总之 Pod 被删除后，Pod 的生命周期就算结束，即使被 Replication Controller 进行重建，那也是新的 Pod，因为 Pod 的 ID 已经发生了变化，所以实际上 Pod 迁移，准确的说法是在新的 Node 上重建 Pod。

4.6.3 生命周期回调函数

Kubernetes 提供了回调函数，在容器的生命周期的特定阶段执行调用，比如容器在停止前希望执行某项操作，就可以注册相应的钩子函数。目前提供的生命周期回调函数如下所示。

- PostStart：在容器创建成功后调用该回调函数。
- PreStop：在容器被终止前调用该回调函数。

钩子函数的实现方式有以下两种。

- Exec

说明

在容器中执行指定的命令。

配置参数

command：需要执行的命令，字符串数组。

示例

```
exec:
```

```
    command:
    - cat
    - /tmp/health
```

- HTTP

说明

发起一个 HTTP 调用请求。

配置参数

path：请求的 URL 路径，可选项。

port：请求的端口，必选项。

host：请求的 IP，可选项，默认是 Pod 的 PodIP。

scheme：请求的协议，可选项，默认是 HTTP。

示例

```
    httpGet:
      host: 192.168.1.1
      path: /notify
      port: 8080
```

现在定义一个 Pod，包含一个 Java 的 Web 应用容器，其中设置了 PostStart 和 PreStop 回调函数。即在容器创建成功后，复制/sample.war 到/app 目录。而在容器被终止之前，发送 HTTP 请求到 http://monitor.com:8080/warning，往监控系统发送一个警告，Pod 的定义如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: javaweb-2
spec:
  containers:
  - image: resouer/sample:v2
    name: war
    lifecycle:
      postStart:
        exec:
          command:
            - "cp"
```

```
        - "/sample.war"
        - "/app"
    preStop:
      httpGet:
        host: monitor.com
        path: /warning
        port: 8080
        scheme: HTTP
```

4.7 自定义检查 Pod

对于 Pod 是否健康，即 Pod 中的容器是否健康，默认情况下只是检查容器是否正常运行。但有时候容器正常运行不代表应用健康，有可能应用的进程已经阻塞住无法正常处理请求，所以为了提供更加健壮的应用，往往需要定制化的健康检查。

除此之外，有的应用启动后需要进行一系列初始化处理，在初始化完成之前应用是无法正常处理请求的。如果应用初始化需要较长时间，而实际上容器创建的时间是可以忽略不计的。默认情况下，Kubernetes 发现容器创建成功并运行，就会认为其准备就绪，真实情况是容器里的应用可能还处于初始化阶段，无法正常接受请求。如果用户访问就会得到错误响应，这不是我们希望看到的情况。同样的，我们需要更加精确的检查机制来判断 Pod 和容器是否准备就绪，从而让 Kubernetes 判断是否分发请求给 Pod。

针对这些需求，Kubernetes 中提供了 Probe 机制，有以下两种类型的 Probe。

- Liveness Probe：用于容器的自定义健康检查，如果 Liveness Probe 检查失败，Kubernetes 将杀死容器，然后根据 Pod 的重启策略来决定是否重启容器。
- Readiness Probe：用于容器的自定义准备状况检查，如果 Readiness Probe 检查失败，Kubernetes 将会把 Pod 从服务代理的分发后端移除，即不会分发请求给该 Pod。

Probe 支持以下三种检查方法。

- ExecAction

说明

在容器中执行指定的命令进行检查，当命令执行成功（返回码为 0），检查成功。

配置参数

command：检查的命令，字符串数组。

示例

```
exec:
  command:
  - cat
  - /tmp/health
```

- TCPSocketAction

说明

对于容器中的指定 TCP 端口进行检查，当 TCP 端口被占用，检查成功。

配置参数

port：检查的 TCP 端口

示例

```
tcpSocket:
  port: 8080
```

- HTTPGetAction

说明

发生一个 HTTP 请求，当返回码介于 200~400 之间时，检查成功。

配置参数

path：请求的 URI 路径，可选项。

port：请求的端口，必选项。

host：请求的 IP，可选项，默认是 Pod 的 PodIP。

scheme：请求的协议，可选项，默认是 HTTP。

示例

```
httpGet:
  path: /healthz
  port: 8080
```

4.7.1 Pod 的健康检查

定义一个 Pod，使用 Liveness Probe 通过 ExecAction 方式检查容器的健康状态，Pod 的定义文件 liveness-exec-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec-pod
  labels:
    test: liveness
spec:
  containers:
  - name: liveness
    image: "ubuntu:14.04"
    command:
    - /bin/sh
    - -c
    - echo ok > /tmp/health; sleep 60; rm -rf /tmp/health; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/health
      initialDelaySeconds: 15
      timeoutSeconds: 1
```

通过定义文件创建 Pod：

```
$ kubectl create -f liveness-exec-pod.yaml
pod "liveness-exec-pod" created
```

Pod 创建之初运行正常：

```
$ kubectl get pod liveness-exec-pod
NAME                READY     STATUS    RESTARTS   AGE
liveness-exec-pod   1/1       Running   0          15s
```

过 1 分钟以后可以看到 Pod 发生了重启：

```
$ kubectl get pod liveness-exec
NAME                READY     STATUS    RESTARTS      AGE
liveness-exec-pod   1/1       Running   1             1m
```

通过查询 Pod 事件可以看到，Liveness Probe 检查失败：

```
$ kubectl describe pod liveness-exec-pod|grep Unhealthy
… Unhealthy   Liveness probe failed: cat: /tmp/health: No such file or directory
```

4.7.2 Pod 的准备状况检查

定义一个 Pod，使用 Readiness Probe 通过 ExecAction 方式检查容器的准备状况，Pod 的定义文件 readiness-exec-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness-exec-pod
spec:
  containers:
  - name: readiness
    image: "ubuntu:14.04"
    command:
    - /bin/sh
    - -c
    - echo ok > /tmp/ready; sleep 60; rm -rf /tmp/ready; sleep 600
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/ready
      initialDelaySeconds: 15
      timeoutSeconds: 1
```

通过定义文件创建 Pod：

```
$ kubectl create -f readiness-exec-pod.yaml
pod "readiness-exec-pod" created
```

Pod 创建之初运行正常，容器全部准备就绪：

```
$ kubectl get pod readiness-exec
NAME                READY   STATUS    RESTARTS   AGE
readiness-exec-pod  1/1     Running   0          26s
```

过 1 分钟以后，发现 Pod 的 READY 数目变为 0：

```
$ kubectl get pod readiness-exec-pod
NAME                READY   STATUS    RESTARTS   AGE
readiness-exec-pod  0/1     Running   0          1m
```

通过查询 Pod 事件可以看到，Readiness Probe 检查失败：

```
$ kubectl describe pod readiness-exec|grep Unhealthy
… Unhealthy   Readiness probe failed: cat: /tmp/ready: No such file or directory
```

4.8 调度 Pod

Pod 的调度指的是 Pod 在创建之后分配到哪一个 Node 上，调度算法分为两个步骤，第一步筛选出符合条件的 Node，第二步选择最优的 Node。

对于所有 Node，首先 Kubernetes 通过一系列过滤函数，去除不符合条件的 Node，当前版本（Kubernetes v1.1.1）支持的过滤函数如下所示。

- NoDiskConflict：检查 Pod 请求的数据卷是否与 Node 上已存在 Pod 挂载的数据卷存在冲突，如果存在冲突，则过滤掉该 Node。
- PodFitsResources：检查 Node 的可用资源（CPU 和内存）是否满足 Pod 的资源请求。
- PodFitsPorts：检查 Pod 设置的 HostPorts 在 Node 上是否已经被其他 Pod 占用。
- PodFitsHost：如果 Pod 设置了 NodeName 属性，则筛选出指定的 Node。
- PodSelectorMatches：如果 Pod 设置了 NodeSelector 属性，则筛选出符合的 Node。
- CheckNodeLabelPresence：检查 Node 是否存在 Kubernetes Scheduler 配置的标签。

筛选出符合条件的 Node 来运行 Pod，如果存在多个符合条件的 Node，那么需要选择出最优的 Node。Kubernetes 中通过一系列优先级函数（Priority Function）来评估出最优 Node。对于每个 Node，优先级函数给出一个分数：0~10（10 表示最优，0 表示最差），而每个优先级函数设置有权重值，Node 的最终分数就是每个优先级函数给出的分数进行加权的和，比如有两个优先级函数 priorityFunc1 和 priorityFunc2，它们的权重值分别是 weight1 和

weight2，那么对于 NodeA 的最终分数是：

```
finalScoreNodeA = (weight1 * priorityFunc1) + (weight2 * priorityFunc2)
```

这样一来，通过最终分数对 Node 进行排序，得分最高的 Node 即最优 Node。如果存在多个 Node 并列第一，则随机选择一个 Node。

当前版本（Kubernetes v1.1.1）的 Kubernetes 提供的优先级函数有如下几个。

- LeastRequestedPriority：优先选择有最多可用资源的 Node。
- CalculateNodeLabelPriority：优先选择含有指定 Label 的 Node。
- BalancedResourceAllocation：优先选择资源使用均衡的 Node。

如何进行 Node 选择呢？

在一些场景下希望 Pod 调度到指定的 Node 上，比如有的 Node 专门用于测试；Pod 在正式上线前，需要先在测试的 Node 上运行，测试完成再发布到生产环境的 Node 上运行。这时候就可以用到 Node Selector，通过 Node 的 Label 进行选择。

查询所有的 Node：

```
$ kubectl get node
NAME          LABELS                                    STATUS    AGE
kube-node-1   kubernetes.io/hostname=kube-node-1   Ready     8d
kube-node-2   kubernetes.io/hostname=kube-node-2   Ready     8d
kube-node-3   kubernetes.io/hostname=kube-node-3   Ready     8d
```

目前共有 3 个 Node，状态都是 Ready，并且有一个默认的 Label kubernetes.io/hostname，然后为 Node kube-node-1 增加新的 Label：

```
$ kubectl label nodes kube-node-1 env=test
node "kube-node-1" labeled
```

在定义 Pod 的时候通过设置 Node Selector（.spec.nodeSelector）来选择 Node，Pod 的定义文件 nginx-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
```

```
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    env: test
```

Pod 创建成功后将会被分配到 Node kube-node-1：

```
$ kubectl get pod nginx -o wide
NAME    READY   STATUS    RESTARTS   AGE     NODE
nginx   1/1     Running   0          9s      kube-node-1
```

除了设置 Node Selector 之外，Pod 还可以通过 Node Name（.spec.nodeName）直接指定 Node：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeName: kube-node-1
```

不过还是建议使用 Node Selector，因为通过 Label 进行选择是一种弱绑定，而直接指定 Node Name 是强绑定，Node 失效时会导致 Pod 无法调度。

4.9 问题定位指南

Pod 是应用的承载体，当 Pod 运行异常的时候，可能是 Kubernetes 系统问题，也可能是应用本身的问题，那么就需要提供足够的信息用于问题定位，Kubernetes 针对 Pod 提供的事件记录、日志查询和远程调试功能进行问题定位。

4.9.1 事件查询

Kubernetes 从 Pod 的创建开始，在 Pod 的生命周期内会产生各种事件信息，比如 Pod 完成调度、下载镜像完成等。在 Pod 运行异常的时候，通过排除相关事件可以了解是否是由于 Kubernetes 的原因导致 Pod 异常。

事件查询可以先查询所有的事件：

```
$ kubectl get event
```

然后再查询 Pod 相关的事件：

```
$ kubectl describe pod my-pod
```

4.9.2 日志查询

日志是一项很重要的信息，可以用来定位问题和显示应用运行状态。Docker 容器可以使用 docker logs 命令查询日志，可以通过 kubectl logs 命令查询 Pod 中容器的日志。

现在要定义一个 Pod，包含两个容器，容器 container1 输出一条日志然后正常退出（exit 0），容器 container2 输出一条日志异常退出（exit 1），并且设置 Pod 的重启策略是 OnFailure，即当容器异常退出时才进行重启，Pod 的定义文件 log-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: log-pod
spec:
  containers:
  - name: container1
    image: ubuntu:14.04
    command:
    - "bash"
    - "-c"
    - "echo \"container1: `date --rfc-3339 ns`\"; exit 0"
  - name: container2
    image: ubuntu:14.04
    command:
    - "bash"
    - "-c"
```

```
    - "echo \"container2: `date --rfc-3339 ns`\"; exit 1"
  restartPolicy: OnFailure
```

通过定义文件创建 Pod：

```
$ kubectl create -f log-pod.yaml
pod "log-pod" created
```

Pod 创建成功后，会重新创建异常退出的容器 container2：

```
$ kubectl get pod log-pod
NAME      READY     STATUS    RESTARTS   AGE
log-pod   0/2       Error     1          19s
```

然后分别查询 Pod 中两个容器的日志：

```
$ kubectl logs log-pod container1
container1: 2015-11-21 14:52:55.622701243+00:00

$ kubectl logs log-pod container2
Pod "log-pod" in namespace "default": container "container2" is in waiting state.
```

因为容器 container2 将会异常退出然后重建，所以将处于异常状态，从而查询不到当前运行日志。但是 kubectl logs 可以查询之前容器（如果存在的话）的日志，这对于问题定位非常有帮助，往往容器停止前的日志价值更高，获取方法只需要加上--previous/-p 参数：

```
$ kubectl logs log-pod container2 –previous
container2: 2015-11-21 14:53:37.377629086+00:00
```

4.9.3 Pod 的临终遗言

前面我们提到过容器停止前的日志价值更高，能够获取最后的错误异常消息、调用栈等，我们可以把这些信息形象地称为临终遗言，临终遗言对于问题定位是很有帮助的。在 Kubernetes 中为 Pod 提供了一个持久化文件，用来保存临终遗言。

Pod 的定义中通过.spec.containers[].terminationMessagePath 指定在容器中的临终遗言日志文件的路径，默认值是/dev/termination-log。这个文件在 Pod 的整个生命周期内都会保存，每次新建一个 Pod，都会在宿主机上创建一个文件，然后挂载到 Pod 的容器中，这些文件不会因为容器的销毁而丢失，所以容器可以把临终遗言写入这个文件，方便问题排错。

现在创建一个 Pod，其中的容器将写入临终遗言，Pod 的定义文件 w-message-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: w-message-pod
spec:
  containers:
  - name: messager
    image: "ubuntu:14.04"
    terminationMessagePath: /dev/termination-log
    command:
    - "bash"
    - "-c"
    - "echo \"`date --rfc-3339 ns` I was going to die\" >> /dev/termination-log;"
```

通过定义文件创建 Pod：

```
$ kubectl create -f w-message-pod.yaml
pod "w-message-pod" created

$ kubectl get pod w-message-pod
NAME           READY     STATUS    RESTARTS   AGE
w-message-pod  0/1       Running   2          3m
```

Pod 运行后，容器将往文件/dev/termination-log 写入临终遗言，然后可以进行查询：

```
$ kubectl get pod w-message-pod \
--template="{{range .status.containerStatuses}}{{.lastState.terminated.message}}{{end}}"
2015-11-21 15:11:57.457833141+00:00 I was going to die
```

4.9.4 远程连接容器

问题定位时往往需要连接到应用的运行环境进行操作，相比于传统的 SSH 方式，Docker 提供了 docker attach 和 docker exec 两个命令可以连接容器进行操作。同样的，Kubernetes 对应地提供了 kubectl attach 和 kubectl exec 两个命令用来远程连接 Pod 中的容器。

其中 attach 命令使用起来不太方便，相比之下，exec 命令则非常强大，我们可以使用 kubectl exec 命令远程连接 Pod 中的容器运行命令（当 Pod 只有一个容器时，不需要指定容器）：

```
$ kubectl exec my-pod -- date
Wed Jan  6 18:19:07 CST 2016
```

或者直接进入 Pod 的容器中：

```
$ kubectl exec -ti my-pod /bin/bash
[root@ my-pod /]#
```

提示

kubectl exec 命令需要在 Kubernetes Node 上安装 nsenter。

第5章

Replication Controller

Replication Controller是Kubernetes中的另一个核心概念，应用托管在Kubernetes之后，Kubernetes需要保证应用能够持续运行，这是Replication Controller的工作内容，它会确保任何时候Kubernetes中有指定数量的Pod副本（或者称为实例）。在此基础上，Replication Controller进一步提供了高级特性，比如弹性伸缩、滚动升级等。本章将详细介绍Replication Controller，其中也会涉及一些相关的内容。

5.1 持续运行的Pod

Kubernetes提供Replication Controller来管理Pod，Replication Controller确保任何时候Kubernetes集群中有指定数量的Pod副本在运行。如果少于指定数量的Pod副本，Replication Controller会启动新的Pod，反之会杀死多余的以保证数量不变。我们通过Replication Controller来创建持续运行的Pod，Replication Controller的定义文件my-nginx-rc.yaml如下所示：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
```

```
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

定义文件中描述了 Replication Controller 的属性和行为，其中的主要要素如下所示。

- apiVersion：声明 Kubernetes 的 API 版本，目前是 v1。
- kind：声明 API 对象的类型，这里的类型是 Replication Controller。
- metadata：设置 Replication Controller 的元数据。
  - name：指定 Replication Controller 的名称，名称必须在 Namespace 内唯一。
- spec：配置 Replication Controller 的具体规格。
  - replicas：设置 Replication Controller 控制的 Pod 的副本数目。
  - selector：指定 Replication Controller 的 Label Selector 来匹配 Pod 的 Label。
- template：设置 Pod 模板，同 Pod 的定义一致。

通过定义文件创建 Replication Controller：

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

查询 Replication Controller：

```
$ kubectl get replicationcontroller my-nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
my-nginx     nginx          nginx      app=nginx   2          14s
```

同时可以查询到 Replication Controller 创建的 Pod：

```
$ kubectl get pod --selector app=nginx --label-columns app
NAME             READY     STATUS    RESTARTS   AGE       APP
my-nginx-14lrs   1/1       Running   0          42s       nginx
my-nginx-cz2gd   1/1       Running   0          42s       nginx
```

因为 Replication Controller 设置 Pod 的副本数目（.spec.replicas）为 2，所以创建出两个 Pod，并且可以看出 Pod 的名称前缀是 Replication Controller 的名称，后缀则是 5 位随机字符串。

现在删除一个 Pod：

```
$ kubectl delete pod my-nginx-14lrs
pod "my-nginx-14lrs" deleted
```

马上可以看到一个新的 Pod 被创建：

```
$ kubectl get pod --selector app=nginx
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-cz2gd   1/1       Running   0          2m
my-nginx-vc43t   1/1       Running   0          29s
```

最后删除 Replication Controller：

```
$ kubectl delete rc my-nginx
replicationcontroller "my-nginx" deleted
```

相应的 Pod 也会被删除：

```
$ kubectl get pods --selector app=nginx
NAME     READY     STATUS    RESTARTS   AGE
```

使用 kubectl delete 命令删除 Replication Controller，默认会删除 Replication Controller 关联的所有 Pod 副本。如果需要保留 Pod 运行，删除 Replication Controller 的时候可以设置参数--cascade=false。

除了 kubectl create 命令之外，也可以通过 kubectl run 命令创建 Replication Controller，下面的命令将创建名称为 my-nginx 的 Replication Controller，创建成功返回 Replication Controller 的信息：

```
$ kubectl run my-nginx --image nginx --replicas 2 --labels app=nginx
replicationcontroller "my-nginx" created
```

5.2 Pod 模板

在实际操作中，相比于直接创建 Pod，一般都是在 Replication Controller 中预先定义 Pod 模板，通过 Replication Controller 来创建 Pod。

Pod 模板是在 Replication Controller 的定义中通过.spec.template 设置的。Pod 模板的定义方法和 Pod 的定义一致，但是有一些需要注意的内容。

- 在 Pod 模板中不需要指定 Pod 的名称，如果指定了，不会报错但是不起作用。因为通过 Pod 模板创建 Pod 的时候会设置 Pod 的.metadata.generateName，然后 Pod 的名称就是.metadata.generateName 拼接上 5 位随机码，这样做的目的是为了通过 Pod 模板创建出来的 Pod 的名称是唯一的。
- Pod 模板中的重启策略必须是 Always，因为 Replication Controller 要保证 Pod 持续运行，必须要求 Pod 总是重启容器，不然谈何持续运行。
- Pod 模板中的 Label 不能为空，否则 Replication Controller 无法同 Pod 模板创建出来的 Pod 进行关联。

下面是一个 Pod 模板示例：

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Replication Controller 使用 Pod 模板创建 Pod，一旦创建成功，Pod 模板和创建的 Pod 就没有关系了。可以修改 Pod 模板但不会对已创建的 Pod 有任何影响，也可以直接更新通过 Replication Controller 创建的 Pod。

提示

在 Kubernetes 中可以单独创建 Pod 模板，PodTemplate 的定义文件 podtemplate.yaml：

```
apiVersion: v1
kind: PodTemplate
metadata:
  name: nginx
template:
  metadata:
```

```
    labels:
      name: nginx
  spec:
    containers:
    - image: nginx
      name: nginx
      ports:
      - containerPort: 80
```

通过定义文件创建 PodTemplate：

```
$ kubectl create -f podtemplate.yaml
podtemplate "nginx" created

$ kubectl get podTemplate nginx
TEMPLATE   CONTAINER(S)   IMAGE(S)   PODLABELS
nginx      nginx          nginx      name=nginx
```

另外，在 Replication Controller 定义中可通过.spec.templateRef 引用 PodTemplate（这部分暂未实现）：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  templateRef:
    name: nginx
```

5.3 Replication Controller 和 Pod 的关联

Replication Controller 和 Pod 的关联就是通过 Label 实现的，Label 机制是 Kubernetes 中很重要的一个设计，通过 Label 进行对象的弱关联，可以灵活地进行分类和选择。就像在社交网络上，对每个人打上不同的标签：职业、年龄、兴趣爱好等，然后系统可以根据标签推送不同的业务，以提供灵活的定制服务。

对于 Pod，需要设置 Label 来进行标识，Label 是一系列的 Key/Value 对，Pod（或者

Pod 模板）的定义中通过.metadata.labels 设置：

```
labels:
  key1: value1
  key2: value2
```

Label 的定义是任意的，但是 Label 必须具有可标识性，比如设置 Pod 的应用名称和版本号等。另外，Label 是不具有唯一性的，为了更准确地标识 Pod，建议为 Pod 设置不同维度的 Label，比如：

- "release" : "stable", "release" : "canary"
- "environment" : "dev","environment" : "qa", "environment" : "production"
- "tier" : "frontend","tier" : "backend", "tier" : "cache"
- "partition" : "customerA","partition" : "customerB"
- "track" : "daily","track" : "weekly"

对于 Replication Controller 来说就是通过 Label Selector 来匹配 Pod 的 Label，从而实现关联关系。在 Replication Controller 的定义中通过.spec.selector 来设置 Label Selector，Replication Controller 的定义文件 my-web-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-web
spec:
  selector:
    app: my-web
  template:
    metadata:
      labels:
        app: my-web
        version: v1
    spec:
      containers:
      - name: my-web
        image: my-web:v1
        ports:
        - containerPort: 80
```

通过定义文件创建 Replication Controller：

```
$ kubectl create -f my-web-rc.yaml
replicationcontroller "my-web" created

$ kubectl get replicationcontroller my-web
CONTROLLER   CONTAINER(S)   IMAGE(S)    SELECTOR     REPLICAS   AGE
my-web       my-web         my-web:v1   app=my-web   1          3s
```

通过 Label 查询 Replication Controller 创建出来的 Pod：

```
$ kubectl get pod --selector app=my-web --label-columns=app
NAME           READY     STATUS    RESTARTS   AGE       APP
my-web-itaxw   1/1       Running   0          27s       my-web
```

可将该 Pod 的 Label app=my-web 修改掉：

```
$ kubectl label pod my-web-itaxw app=debug --overwrite
pod "my-web-itaxw" labeled
```

通过查询可以看到马上有新的 Pod 被创建出来，因为 Replication Controller 正是通过 Label 关联 Pod，Pod 的 Label 被修改掉，对 Replication Controller 而言相当于减少一个关联的 Pod，自然就会创建新的 Pod。

```
$ kubectl get pod --selector app --label-columns=app
NAME           READY     STATUS    RESTARTS   AGE       APP
my-web-itaxw   1/1       Running   0          1m        debug
my-web-p41ln   1/1       Running   0          16s       my-web
```

对于修改标签的 Pod 来说，相当于脱离 Replication Controller 的控制，这种方法适合对应用进行调试（或者数据备份），相当于有一个脱离控制的 Pod 可以进行调试，最后可以删除这个 Pod：

```
$ kubectl delete pod my-web-itaxw
pod "my-web-itaxw" deleted

$ kubectl get pod --selector app --label-columns=app
NAME           READY     STATUS    RESTARTS   AGE       APP
my-web-p41ln   1/1       Running   0          1m        my-web
```

如果试图创建 Label 为 app=my-web 的 Pod，会发现 Pod 是创建不出来的，因为这个 Pod 和 Replication Controller 的 Label Selector 匹配了，那么 Replication Controller 会删除这

个 Pod 来保证 Pod 的副本数不多也不少。所以当多个 Replication Controller 的 Label Selector 一样且设置的 Pod 副本不一样的时候，会产生冲突。

5.4 弹性伸缩

弹性伸缩是指适应负载变化，以弹性可伸缩方式提供资源，特别是在虚拟化支持下，提高资源的利用率和用户满意度，较好地解决了资源利用率和应用系统之间的矛盾，是云计算领域的关键能力。想象一下现实世界，比如大型超市的出口，高峰时期人流量大的时候，适时增加结算柜台，而当人流量少的时候减少结算柜台。

同样反映到 Kubernetes 中，可根据负载的高低动态调整 Pod 的副本数，目前 Kubernetes 提供了 API 接口实现 Pod 的弹性伸缩。当然，Pod 的副本数本来就是通过 Replication Controller 进行控制，所以 Pod 的弹性伸缩就是修改 Replication Controller 的 Pod 副本数，可以通过 kubectl scale 命令来完成。

首先创建 Replication Controller，设置的 Pod 副本数为 1，Replication Controller 的定义文件 my-nginx-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
      restartPolicy: Never
```

通过定义文件创建 Replication Controller：

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

Replication Controller 创建成功后，创建出指定数目的 Pod：

```
$ kubectl get replicationcontroller my-nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
my-nginx     nginx          nginx      app=nginx   1          1m

$ kubectl get pod --selector app=nginx
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-sb6my   1/1       Running   0          1m
```

扩容 Pod 的副本数目到 3：

```
$ kubectl scale replicationcontroller my-nginx --replicas=3
replicationcontroller "my-nginx" scaled

$ kubectl get replicationcontroller --selector app=nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
my-nginx     nginx          nginx      app=nginx   3          2m

$ kubectl get pod --selector app=nginx
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-2hzis   1/1       Running   0          49s
my-nginx-7b66n   1/1       Running   0          49s
my-nginx-sb6my   1/1       Running   0          2m
```

缩容 Pod 的副本数到 1：

```
$ kubectl scale replicationcontroller my-nginx --replicas=1
replicationcontroller "my-nginx" scaled

$ kubectl get replicationcontroller --selector app=nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
my-nginx     nginx          nginx      app=nginx   1          3m

$ kubectl get pod --selector app=nginx
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-7b66n   1/1       Running   0          1m
```

kubectl scale 如果设置--current-replicas 参数，会先检查当前的 Pod 的副本数是否匹配，不匹配的话会报错：

```
$ kubectl scale rc my-nginx --current-replicas=2 --replicas=1
Expected replicas to be 2, was 1
```

另外，也可以把 Pod 的副本数目设置为 0，即删除 Replication Controller 关联的所有 Pod：

```
$ kubectl scale replicationcontroller my-nginx --replicas=0
replicationcontroller "my-nginx" scaled

$ kubectl get replicationcontroller --selector app=nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
my-nginx     nginx          nginx      app=nginx   0          4m

$ kubectl get pod --selector app=nginx
NAME      READY     STATUS    RESTARTS   AGE
```

5.5 自动伸缩

通过 Replication Controller 可以非常方便地实现 Pod 的弹性伸缩，在此基础上，只要有平台监控支持，就可以实现自动伸缩的功能，即基于 Pod 的资源使用情况，根据配置的策略自动调整 Pod 的副本数。

在 Kubernetes 中通过 Horizontal Pod Autoscaler 来实现 Pod 的自动伸缩，Horizontal Pod Autoscaler 同 Replication Controller 是一一对应的，

Horizontal Pod Autoscaler 将定时从平台监控系统中获取 Replication Controller 关联 Pod 的整体资源使用情况。当策略匹配的时候，通过 Replication Controller 来调整 Pod 的副本数，实现自动伸缩。

> **提示**
>
> 在当前版本（Kubernetes v1.1.1）中，Horizontal Pod Autoscaler 处于 Beta 测试阶段，目前策略只支持根据 CPU 的使用情况进行关联，并且 Kubernetes 需要安装平台监控（可参考 2.3.2 节）。

通过 kubectl run 创建 Replication Controller，将运行一个 Nginx Pod：

```
$ kubectl run nginx --image=nginx --labels app=nginx --requests cpu=200m
replicationcontroller "nginx" created

$ kubectl get replicationcontroller nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
nginx        nginx           nginx      app=nginx   1          1m

$ kubectl get pod --selector app=nginx
NAME          READY     STATUS    RESTARTS   AGE
nginx-dqsvi   1/1       Running   0          1m
```

通过 kubectl expose 创建 Service，代理 Nginx Pod，然后就可以通过 http://10.254.133.192 访问 Nginx Pod：

```
$ kubectl expose replicationcontroller nginx --port=80
service "nginx" exposed

$ kubectl get service nginx
NAME      CLUSTER_IP       EXTERNAL_IP   PORT(S)   SELECTOR    AGE
nginx     10.254.133.192   <none>        80/TCP    app=nginx   5s
```

现在将创建 Horizontal Pod Autoscaler 来实现 Nginx Pod 的弹性伸缩，Horizontal Pod Autoscaler 定义文件 nginx-hpa.yaml：

```
apiVersion: extensions/v1beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  scaleRef:
    kind: ReplicationController
    name: nginx
    subresource: scale
  minReplicas: 1
  maxReplicas: 10
  cpuUtilization:
    targetPercentage: 50
```

在此 Horizontal Pod Autoscaler 中通过.spec.scaleRef 指定对应的 ReplicationController，.spec.minReplicas 和.spec.maxReplicas 分别设定 Pod 伸缩的最小和最大副本数。另外设置了自动

伸缩策略：当所有关联 Pod 的 CPU 平均使用率超过 50%的时候进行扩容，而少于 50%的时候，进行缩容。

现在通过定义文件创建 Horizontal Pod Autoscaler：

```
$ kubectl create -f nginx-hpa.yaml
horizontalpodautoscaler "nginx" created
```

提示

可以通过 kubectl autoscale 创建 Horizontal Pod Autoscaler：

```
$ kubectl autoscale rc nginx --min=1 --max=10 --cpu-percent=50
replicationcontroller "nginx" autoscaled
```

创建成功后查询 Horizontal Pod Autoscaler：

```
$ kubectl get horizontalpodautoscaler nginx
NAME     REFERENCE                          TARGET    CURRENT   MINPODS   MAXPODS   AGE
nginx    ReplicationController/nginx/scale  50%       0%        1         10        1d
```

因为没有访问量，所以当前关联 Pod 的 CPU 平使用率为 0%，我们可以使用工具增加访问量。当访问量增加的时候，查询 Horizontal Pod Autoscaler，观察变化：

```
$ kubectl get horizontalpodautoscaler nginx
NAME     REFERENCE                          TARGET    CURRENT   MINPODS   MAXPODS   AGE
nginx    ReplicationController/nginx/scale  50%       60%       1         10        1d
```

当 CPU 平均使用率超过 50%的时候，可以发现对应的 Replication Controller 的 Pod 副本数变为 2，实现了扩容。

```
$ kubectl get horizontalpodautoscaler nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
nginx        nginx          nginx      app=nginx   2          10m
```

当我们停止访问，CPU 平均使用率降到 50%以下，Replication Controller 的 Pod 副本数变为 1，即实现了缩容。

```
$ kubectl get horizontalpodautoscaler nginx
NAME     REFERENCE                          TARGET    CURRENT   MINPODS   MAXPODS   AGE
nginx    ReplicationController/nginx/scale  50%       43%       1         10        1d
```

```
$ kubectl get replicationcontroller nginx
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS   AGE
nginx        nginx          nginx      app=nginx   1          12m
```

5.6 滚动升级

滚动升级是一种平滑过渡的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始升级的时候就可以及时发现、调整问题，以保证问题影响度不会扩大。

在 Kubernetes 中支持滚动升级，现在我们通过一个例子演示应用从 V1 版本滚动升级到 V2 版本。首先创建 V1 版本的 Replication Controller，V1 版本的 Replication Controller 的定义文件 my-web-v1-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-web-v1
spec:
  selector:
    app: my-web
    version: v1
  template:
    metadata:
      labels:
        app: my-web
        version: v1
    spec:
      containers:
      - name: my-web
        image: my-web:v1
        ports:
        - containerPort: 80
          protocol: TCP
```

通过定义文件创建 Replication Controller：

```
$ kubectl create -f my-web-v1-rc.yaml
replicationcontroller "my-web-v1" created
```

```
$ kubectl get replicationcontroller my-web-v1
CONTROLLER   CONTAINER(S)   IMAGE(S)    SELECTOR                  REPLICAS   AGE
my-web-v1    my-web         my-web:v1   app=my-web, version=v1    1          10s
```

然后扩容 Replication Controller 的 Pod 副本数目到 4：

```
$ kubectl scale rc my-web-v1 --replicas=4
replicationcontroller "my-web-v1" scaled

$ kubectl get pods --selector app=my-web
NAME              READY     STATUS    RESTARTS   AGE
my-web-v1-3dd6v   1/1       Running   0          9s
my-web-v1-8893c   1/1       Running   0          57s
my-web-v1-il6ii   1/1       Running   0          8s
my-web-v1-rzjp5   1/1       Running   0          9s
```

现在需要应用从 V1 版本升级到 V2 版本，V2 版本的 Replication Controller 定义文件 my-web-v2-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-web-v2
spec:
  selector:
    app: my-web
    version: v2
  template:
    metadata:
      labels:
        app: my-web
        version: v2
    spec:
      containers:
      - name: my-web
        image: my-web:v2
        ports:
        - containerPort: 80
          protocol: TCP
```

开始滚动升级：

```
$ kubectl rolling-update my-web-v1 -f my-web-v2-rc.yaml --update-period=10s
Created my-web-v2
Scaling up my-web-v2 from 0 to 4, scaling down my-web-v1 from 4 to 0 (keep 4 pods available,
don't exceed 5 pods)
Scaling my-web-v2 up to 1
Scaling my-web-v1 down to 3
Scaling my-web-v2 up to 2
Scaling my-web-v1 down to 2
Scaling my-web-v2 up to 3
Scaling my-web-v1 down to 1
Scaling my-web-v2 up to 4
Scaling my-web-v1 down to 0
Update succeeded. Deleting my-web-v1
replicationcontroller "my-web-v1" rolling updated to "my-web-v2"
```

升级开始后，首先根据提供的定义文件创建 V2 版本的 Replication Controller，然后每隔 10s（通过 kubectl rolling-update 的参数--update-period 设置）逐步增加 V2 版本的 Replication Controller 的 Pod 副本数，逐步减少 V1 版本的 Replication Controller 的 Pod 副本数。升级完成后删除 V1 版本的 Replication Controller，保留 V2 版本的 Replication Controller，即实现滚动升级。

```
Updating my-web-v1 replicas: 3, my-web-v2 replicas: 1
Updating my-web-v1 replicas: 2, my-web-v2 replicas: 2
Updating my-web-v1 replicas: 1, my-web-v2 replicas: 3
Updating my-web-v1 replicas: 0, my-web-v2 replicas: 4
```

升级期间通过查询 Pod 可知，V2 版本的 Pod 正在逐渐替换 V1 版本的 Pod，当然这是通过调整 Replication Controller 的副本数目来控制的，下面显示的是升级过程中的一个阶段的状况：

```
$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER   CONTAINER(S)   IMAGE(S)    SELECTOR                    REPLICAS AGE
my-web-v1    my-web         my-web:v1   app=my-web, version=v1      2        2m
my-web-v2    my-web         my-web:v2   app=my-web, version=v2      2        48s

$ kubectl get pod --selector app=my-web
NAME            READY   STATUS      RESTARTS      AGE
my-web-v1-3dd6v  1/1    Running     0             1m
```

```
my-web-v1-8893c   1/1     Running     0          2m
my-web-v2-6cg74   1/1     Running     0          19s
my-web-v2-b4qaz   1/1     Running     0          45s
```

待升级完成，即 V2 版本的 Pod 完全替换 V1 版本的 Pod，同时 V1 版本的 Replication Controller 也被 V2 版本的 Replication Controller 替换：

```
$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER   CONTAINER(S)   IMAGE(S)      SELECTOR                  REPLICAS
my-web-v2    my-web         my-web:v2     app=my-web, version=v2    4

$ kubectl get pod --selector app=my-web
NAME              READY     STATUS    RESTARTS   AGE
my-web-v2-785ok   1/1       Running   0          35s
my-web-v2-bg8ur   1/1       Running   0          1m
my-web-v2-qr33c   1/1       Running   0          1m
my-web-v2-y2es3   1/1       Running   0          57s
```

如果在升级过程中，发生了错误中途退出的时候，可以选择继续升级。Kubernetes 能够智能地判断出升级中断之前的阶段，然后紧接着继续执行升级。另外，也可以进行回退，命令如下：

```
$ kubectl rolling-update my-web-v1 -f my-web-v2-rc.yaml --update-period=10s --rollback
Setting "my-web-v1" replicas to 4
Continuing update with existing controller my-web-v1.
Scaling up my-web-v1 from 3 to 4, scaling down my-web-v2 from 2 to 0 (keep 2 pods available,
don't exceed 3 pods)
Scaling my-web-v2 down to 0
Scaling my-web-v1 up to 4
Update succeeded. Deleting my-web-v2
```

回退的方式实际上就是升级的逆操作，逐步增加 V1 版本的 Replication Controller 的副本数，逐步减少 V2 版本的 Replication Controller 的副本数。

5.7 Deployment

Kubernetes 提供了一种更加简单的更新 Replication Controller 和 Pod 的机制，叫作 Deployment。

> **提示**
> 在当前版本（Kubernetes v1.1.1）中，Deployment 处于 beta 测试阶段，需要 Kubernetes API Server 的启动参数设置--runtime-config=extensions/v1beta1/deployments=true 开启 Deployment 支持。

我们创建一个 Deployment，Deployment 的定义文件 my-web-v1-deployment.yaml：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-web-deployment
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: my-web
    spec:
      containers:
      - name: my-web
        image: my-web:v1
        ports:
        - containerPort: 80
          protocol: TCP
```

Deployment 的定义方法与 Replication Controller 类似，包括 Pod 副本数和 Pod 副本的设置，这些定义将会作用于 Deployment 创建的 Replication Controller。

通过定义文件创建 Deployment：

```
$ kubectl create -f my-web-v1-deployment.yaml --validate=false
deployment "my-web-deployment" created
```

创建成功后可以查询到 Deployment 和其创建的 Replication Controller：

```
$ kubectl get deployment my-web-deployment
NAME                UPDATEDREPLICAS    AGE
my-web-deployment   0/4                32s

$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER              CONTAINER(S)   IMAGE(S)    SELECTOR                                                          REPLICAS   AGE
deploymentrc-3379117081 my-web         my-web:v1   app=my-web, deployment.kubernetes.io/podTemplateHash=3379117081   0          21s
```

Deployment 创建出来的 Replication Controller 的名称是 deploymentrc-3379117081，Replication Controller 使用的是 Deployment 定义中的 Pod 模板。

Deployment 给 Replication Controller 添加了一个 Label deployment.kubernetes.io/podTemplateHash=3379117081，其中 Label 的 Value 是 3379117081，这是由 Pod 模板计算出的 Hash 值，Label 的 Key 是由 Deployment 中的.spec.uniqueLabelKey 指定的，默认是 deployment.kubernetes.io/podTemplateHash，如果为空，则不添加这个 Label。

Deployment 中通过.spec.replicas 指定了预期的 Pod 副本数，不过在刚创建的时候，初始值是 0。

过一段时间后，Deployment 将会设置 Replication Controller 的 Pod 副本数为 4，相应地创建出了 4 个 Pod：

```
$ kubectl get deployment my-web-deployment
NAME                UPDATEDREPLICAS    AGE
my-web-deployment   4/4                50s

$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER              CONTAINER(S)   IMAGE(S)   SELECTOR                                                          REPLICAS   AGE
deploymentrc-3379117081   my-web       my-web:v1  app=my-web, deployment.kubernetes.io/podTemplateHash=3379117081   4          1m

$ kubectl get pods --selector app=my-web --label-columns deployment.kubernetes.io/
podTemplateHash
NAME                              READY     STATUS    RESTARTS   AGE       PODTEMPLATEHASH
deploymentrc-3379117081-5ghmj     1/1       Running   0          1m        3379117081
deploymentrc-3379117081-g4a0i     1/1       Running   0          1m        3379117081
deploymentrc-3379117081-orckb     1/1       Running   0          1m        3379117081
deploymentrc-3379117081-r69kt     1/1       Running   0          1m        3379117081
```

现在我们需要更新应用，镜像 my-web:v1 升级到 my-web:v2，为此，新的 Deployment 定义文件 my-web-v2-deployment.yaml 如下所示：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-web-deployment
spec:
  replicas: 4
  template:
```

```
  metadata:
    labels:
      app: my-web
  spec:
    containers:
    - name: my-web
      image: my-web:v2
      ports:
      - containerPort: 80
        protocol: TCP
```

使用新的定义文件更新 Deployment：

```
$ kubectl apply -f my-web-v2-deployment.yaml --validate=false
deployment "my-web-deployment" configured
```

更新生效后，可以查询到 Deployment 创建了新的 Replication Controller：

```
$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER          CONTAINER(S)  IMAGE(S)   SELECTOR                                                                    REPLICAS  AGE
deploymentrc-3379117081  my-web   my-web:v1  app=my-web, deployment.kubernetes.io/podTemplateHash=3379117081  4         2m
deploymentrc-3445177370  my-web   my-web:v2  app=my-web, deployment.kubernetes.io/podTemplateHash=3445177370  0         6s
```

新创建出的 Replication Controller 名称为 deploymentrc-3445177370，Label 设置为 deployment.kubernetes.io/podTemplateHash=3445177370，3445177370 是由新的 Pod 模板计算出的 Hash 值，新的 Replication Controller 使用镜像 my-web:v2，初始的 Pod 副本数为 0。

紧接着 Deployment 就会控制新旧 Replication Controller，实现 Pod 的升级：

```
$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER          CONTAINER(S)  IMAGE(S)   SELECTOR                                                                    REPLICAS  AGE
deploymentrc-3379117081  my-web   my-web:v1  app=my-web, deployment.kubernetes.io/podTemplateHash=3379117081  0         4m
deploymentrc-3445177370  my-web   my-web:v2  app=my-web, deployment.kubernetes.io/podTemplateHash=3445177370  4         2m
```

```
$ kubectl get pod --selector app=my-web --label-columns deployment.kubernetes.io/
podTemplateHash
NAME                                READY     STATUS    RESTARTS   AGE       PODTEMPLATEHASH
deploymentrc-3445177370-4ej3a       1/1       Running   0          2m        3445177370
deploymentrc-3445177370-i327a       1/1       Running   0          1m        3445177370
deploymentrc-3445177370-reeqh       1/1       Running   0          46s       3445177370
deploymentrc-3445177370-zhopb       1/1       Running   0          1m        3445177370
```

默认情况下，Deployment 采取的是滚动升级方式，同使用 kubectl rolling-update 是一致的，即逐步增加新的 Replication Controller 的副本数，逐步减少旧的 Replication Controller 的副本数，通过查询 Deployment 的事件可以看到具体变化情况：

```
$ kubectl describe deployment my-web-deployment
...
Scaled up rc deploymentrc-3379117081 to 4
Scaled up rc deploymentrc-3445177370 to 1
Scaled down rc deploymentrc-3379117081 to 2
Scaled up rc deploymentrc-3445177370 to 3
Scaled down rc deploymentrc-3379117081 to 0
Scaled up rc deploymentrc-3445177370 to 4
```

在 Deployment 中可以配置升级的策略，通过.spec.strategy.type 指定升级类型，包括 Recreate 和 RollingUpdate。

- Recreate：直接升级，即删除所有旧的 Pod，然后创建新的 Pod，当前版本（Kubernetes v1.1.1）暂未实现。
- RollingUpdate：滚动升级，支持参数配置，如表 5-1 所示。

表 5-1 RollingUpdate 策略参数

参数	说明
.spec.strategy.rollingUpdate.maxUnavailable	允许的最大失效Pod数值，可选配置，值可以是绝对值（5）或者是比例（10%），默认值是1。比如maxUnavailable是30%，升级开始后，旧的Replication Controller可以立即缩容30%的Pod，当新的Replication Controller创建出Pod以后，旧的Replication Controller可以进一步缩容，整个升级过程至少需要保证70%的Pod可用
.spec.strategy.rollingUpdate.maxSurge	允许超过指定Pod数目的最大数值，可选配置，值可以是绝对值（5）或者是比例（10%），默认值是1。比如maxSurge是30%，升级开始后，新的Replication Controller可以立即扩容30%，旧的Replication Controller删除Pod之后，新的Replication Controller可以继续扩容，在整个过程中，新旧Pod的总数目不可以超过指定数目的130%

5.8 一次性任务的 Pod

从程序运行形态上来区分，我们可以将 Pod 分为两类：长时运行服务和一次性任务。大部分应用比如 Nginx、Redis 等都是长时运行服务，另外也存在一次性任务的场景，比如执行冒烟测试、数据计算等。

Replication Controller 创建的 Pod 都是长时运行服务，相应的，Kubernetes 提供了另一种机制，Job 来管理一次性任务的 Pod。

> **提示：**
> 在当前版本（Kubernetes v1.1.1）中，Job 处于 Beta 测试阶段。

我们现在使用 Job 来计算圆周率，Job 的定义文件 pi-job.yaml：

```
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: pi
spec:
  completions: 1
  parallelism: 1
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      name: pi
      labels:
        app: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Job 同样是通过.spec.template 配置 Pod 的模板的，因为是一次性任务 Pod，所以 Pod 的重启策略只能是 Never 或者 OnFailure。

Job 可以控制一次性任务 Pod 的完成次数（.spec.completions）和并发执行数（.spec.parallelism），当 Pod 成功执行指定次数后即认为 Job 执行完毕。

通过定义文件创建 Job：

```
$ kubectl create -f pi-job.yaml
job "pi" created

$ kubectl get job
JOB       CONTAINER(S)  IMAGE(S)   SELECTOR      SUCCESSFUL
pi        pi            perl       app in (pi)   0
```

Job 创建成功后，将会创建运行 Pod：

```
$ kubectl get pod --selector app=pi
NAME       READY     STATUS    RESTARTS   AGE
pi-8117p   1/1       Running   0          6s
```

Pod 是一次性任务，计算出圆周率就终止，我们可以查询到 Pod 输出的圆周率：

```
$ kubectl logs pi-8117p
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862
80340...
```

在一次性任务 Pod 执行完后，显示 Job 已经成功执行 1 次，即完成任务：

```
$ kubectl get job pi
JOB       CONTAINER(S)  IMAGE(S)   SELECTOR      SUCCESSFUL
pi        pi            perl       app in (pi)   1
```

第 6 章

Service

为了适应快速的业务需求，微服务架构已经逐渐成为主流，微服务架构的应用需要有非常好的服务编排支持。Kubernetes 中的核心要素 Service 便提供了一套简化的服务代理和发现机制，天然适应微服务架构，任何应用都可以非常轻易地运行在 Kubernetes 中而无须对架构进行改动。本章将阐述 Service 的基本概念和功能细节，最后介绍如何将 Service 发布给外部网络的方法。

6.1 Service 代理 Pod

在 Kubernetes 中，在受到 Replication Controller 支配的时候，Pod 副本是变化的，比如发生迁移（准确说是 Pod 的重建）或者伸缩的时候。这对于 Pod 的访问者来说就是一种负担，访问者需要能够发现这些 Pod 副本，并且感知 Pod 副本的变化以便及时进行更新。

Kubernetes 中的 Service 是一种抽象概念，它定义了一个 Pod 逻辑集合以及访问它们的策略，Service 同 Pod 的关联同样是居于 Label 来完成的。Service 的目标是提供一种桥梁，它会为访问者提供一个固定访问地址，用于在访问时重定向到相应的后端，这使得非 Kubernetes 原生应用程序，在无须为 Kubernetes 编写特定代码的前提下，轻松访问后端。

我们现在定义一个 Service 来代理 Pod，首先创建 Replication Controller，Replication Controller 的定义文件 my-nginx-rc.yaml：

```
apiVersion: v1
```

```
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

通过定义文件创建 Replication Controller：

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

Replication Controller 创建出 3 个 Pod 副本：

```
$ kubectl get pod --selector app=nginx
NAME              READY     STATUS    RESTARTS   AGE
my-nginx-2ywz1    1/1       Running   0          1m
my-nginx-arjq1    1/1       Running   0          1m
my-nginx-tok0x    1/1       Running   0          1m
```

然后创建 Service，Service 的定义文件 my-nginx-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
  - port: 80
```

```
    targetPort: 80
    protocol: TCP
```

定义文件中描述了 Service 的属性和行为，其中的主要要素如下所示

- apiVersion：声明 Kubernetes 的 API 版本，目前是 v1。
- kind：声明 API 对象的类型，这里类型是 Service。
- metadata：设置 Service 的元数据。
  - name：指定 Service 的名称，名称必须在 Namespace 内唯一。
- spec：配置 Service 的具体规格。
  - selector：指定 Service 的 Label Selector 来匹配 Pod 的 Label。
  - ports：设置 Service 的端口转发规则。

通过定义文件创建 Service：

```
$ kubectl create -f my-nginx-service.yaml
service "my-nginx" created
```

提示

除了 kubectl create 之外，也可以通过 kubectl expose 创建 Service：

```
$ kubectl expose replicationcontroller my-nginx --name=my-nginx --port=80 --target-
port=80
service "my-nginx" exposed
```

创建成功后可以查询 Service：

```
$ kubectl get service my-nginx
NAME       CLUSTER_IP      EXTERNAL_IP   PORT(S)   SELECTOR   AGE
my-nginx   10.254.226.117  <none>        80/TCP    app=nginx  43s
```

Service 同 Replication Controller 一样都是通过 Label 来关联 Pod 的，Service 的定义中指定了 Label Selector 为 app=nginx，那么就会关联前面运行的 3 个 Pod。Service 会将关联到的 Pod 作为 Service 分发请求的后端，可以查询 Service：

```
$ kubectl describe service my-nginx
Name:              my-nginx
Namespace:         default
Labels:            <none>
Selector:          app=nginx
Type:              ClusterIP
```

```
IP:                10.254.226.117
Port:              <unnamed>80/TCP
Endpoints:         10.0.62.68:80, 10.0.62.69:80, 10.0.62.70:80
Session Affinity:  None
No events.
```

可以看到 Service 的 Endpoints 属性包含了 3 个 IP: 10.0.62.68、10.0.62.69、10.0.62.70，实际上这就是 Service 关联的 3 个 Pod 的 PodIP：

```
$ kubectl get pod --selector app=nginx -o yaml|grep podIP
  podIP: 10.0.62.70
  podIP: 10.0.62.68
  podIP: 10.0.62.69
```

我们看到，Service 的 IP 属性显示为 10.254.226.117，实际上这是 Kubernetes 分配给 Service 的一个虚拟 IP。通过访问 Service 的虚拟 IP，Kubernetes 会转发请求到后端 Pod。另外，Service 的端口转发规则显示 Service 的 80/TCP 端口（通过 spec.ports[0].port 指定）转发到后端的 80 端口（通过 spec.ports[0].targetPort 指定），比如访问 10.254.226.117:80 的请求会被转发到后端 10.0.62.68:80、10.0.62.69:80、10.0.62.70:80：

```
$ curl 10.254.226.117:80
...Thank you for using nginx...
```

而当 Pod 发生变化的时候，Service 会及时更新，比如将 Pod 的副本数目减少至 1：

```
$ kubectl scale replicationcontroller my-nginx --replicas=1
replicationcontroller "my-nginx" scaled

$ kubectl get pod --selector app=nginx
NAME           READY     STATUS    RESTARTS   AGE
my-nginx-tok0x  1/1      Running   0          7m
```

Service 相应地进行了更新：

```
$ kubectl describe service my-nginx
Name:              my-nginx
Namespace:         default
Labels:            <none>
Selector:          app=nginx
Type:              ClusterIP
IP:                10.254.226.117
Port:              <unnamed>80/TCP
```

```
Endpoints:         10.0.62.69:80
Session Affinity:  None
No events.
```

这样一来，Service 就可以作为 Pod 的访问入口，起到代理服务器的作用，而对于访问者来说，通过 Service 进行访问，无须直接感知 Pod。

6.2 Service 的虚拟 IP

Kubernetes 分配给 Service 一个固定 IP，这是一个虚拟 IP（也称为 ClusterIP），并不是一个真实存在的 IP，而是由 Kubernetes 虚拟出来的。虚拟 IP 的范围通过 Kubernetes API Server 的启动参数--service-cluster-ip-range=10.254.0.0/16 配置，查询 Service：

```
$ kubectl get service
NAME         CLUSTER_IP       EXTERNAL_IP   PORT(S)        SELECTOR      AGE
kubernetes   10.254.0.1       <none>        443/TCP        <none>        6d
my-nginx     10.254.226.117   <none>        80/TCP         app=nginx     43s
```

可以查询到两个 Service，其中第一个 Service 是由 Kubernetes 默认创建的，它代表着 Kubernetes API Server。两个 Service 的虚拟 IP 都属于 10.254.0.0/16 范围，在 Service 定义中可以通过.spec.clusterIP 指定虚拟 IP，指定的 IP 必须在指定范围内，并且该虚拟 IP 未被分配使用：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  clusterIP: 10.254.249.161
```

如果 Service 设置.spec.clusterIP 为 None，表示不给 Service 分配虚拟 IP，我们称为

Headless Service：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  clusterIP: None
```

虚拟 IP 属于 Kubernetes 内部的虚拟网络，外部是寻址不到的。在 Kubernetes 系统中，实际上是由 Kubernetes Proxy 组件负责实现虚拟 IP 路由和转发的，所以在 Kubernetes Node 中我们都运行了 Kubernetes Proxy，从而在容器覆盖网络之上又实现了 Kubernetes 层级的虚拟转发网络。

6.3 服务代理

在逻辑层面上，Service 可以被认为是真实应用的抽象，每一个 Service 关联着一系列的 Pod。在物理层面上，Service 又是真实应用的代理服务器，对外表现为一个单一访问入口，通过 Kubernetes Proxy 转发请求到 Service 关联的 Pod。

Service 同样是根据 Label Selector 来筛选 Pod 进行关联的，实际上 Kubernetes 在 Service 和 Pod 之间通过 Endpoints 衔接，Endpoints 同 Service 关联的 Pod 相对应，可以认为是 Service 的服务代理后端，Kubernetes 会根据 Service 关联到的 Pod 的 PodIP 信息组合成一个 Endpoints。

我们现在创建一个 Service，定义文件 my-nginx-service.yaml：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
```

```
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  - name: https
    port: 443
    targetPort: 443
    protocol: TCP
```

Service 的定义中设置了两个端口转发规则。当 Service 只配置一个端口的时候，端口的名称是可选项，而当 Service 配置多个端口的时候，每个端口的 name 就是必选项。

通过定义文件创建 Service：

```
$ kubectl create -f my-nginx-service.yaml
service "my-nginx" created

$ kubectl get service my-nginx
NAME       CLUSTER_IP      EXTERNAL_IP   PORT(S)          SELECTOR    AGE
my-nginx   10.254.181.218  <none>        80/TCP, 443/TCP  app=nginx   3s
```

Service 将通过 Label app=nginx 关联到 1 个 Pod：

```
$ kubectl get pod --selector app=nginx
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-4s97i   1/1       Running   0          48s
```

Kubernetes 创建 Service 的同时，会自动创建跟 Service 同名的 Endpoints：

```
$ kubectl get endpoints my-nginx -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2015-11-28T03:35:52Z
  name: my-nginx
  namespace: default
  resourceVersion: "224471"
  selfLink: /api/v1/namespaces/default/endpoints/my-nginx
  uid: 1ff974f4-9581-11e5-b92e-005056817c3e
```

```
subsets:
- addresses:
  - ip: 10.0.62.71
    targetRef:
      kind: Pod
      name: my-nginx-4s97i
      namespace: default
      resourceVersion: "224453"
      uid: 33e8b6e0-9581-11e5-b92e-005056817c3e
  ports:
  - name: http
    port: 80
    protocol: TCP
  - name: https
    port: 443
    protocol: TCP
```

可以看到 Endpoints 包含 Service 关联到的 Pod 的 PodIP，Service 根据端口对应 Endpoints 的相应端口：

```
$ kubectl describe service my-nginx
Name:           my-nginx
Namespace:      default
Labels:         <none>
Selector:       app=nginx
Type:           ClusterIP
IP:             10.254.181.218
Port:           http 80/TCP
Endpoints:      10.0.62.71:80
Port:           https   443/TCP
Endpoints:      10.0.62.71:443
Session Affinity: None
No events.
```

Service 不仅可以代理 Pod，还可以代理任意其他后端，比如运行在 Kubernetes 外部的服务。假设现在要使用一个 Service 代理外部 MySQL 服务，不用设置 Service 的 Label Selector，Service 的定义文件 mysql-service.yaml：

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: mysql
spec:
  ports:
  - port: 3036
    targetPort: 3036
    protocol: TCP
```

同时定义跟 Service 同名的 Endpoints，Endpoints 中设置了 MySQL 的 IP：192.168.3.180，Endpoints 的定义文件 mysql-endpoints.yaml：

```
apiVersion: v1
kind: Endpoints
metadata:
  name: mysql
subsets:
- addresses:
  - ip: 192.168.3.180
  ports:
  - port: 3036
    protocol: TCP
```

通过定义文件创建 Service 和 Endpoints：

```
$ kubectl create -f mysql-service.yaml -f mysql-endpoints.yaml
service "mysql" created
endpoints "mysql" created
```

可以查询到 Service 将指向我们自定义的 Endpoints：

```
$ kubectl get endpoints mysql
NAME      ENDPOINTS           AGE
mysql     192.168.3.180:3036  23s

$ kubectl describe service mysql
Name:           mysql
Namespace:      default
Labels:         <none>
Selector:       <none>
Type:           ClusterIP
IP:             10.254.223.0
Port:           <unnamed>3036/TCP
```

```
Endpoints:    192.168.3.180:3036
Session Affinity: None
No events.
```

当 Service 的 Endpoints 包含多个 IP 的时候，即服务代理存在多个后端，将进行请求的负载均衡，默认的负载均衡策略是轮询或者随机（根据 Kubernetes Proxy 的模式决定）。Service 支持基于源 IP 地址的会话保持，通过.spec.sessionAffinity 设置为 ClientIP：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  sessionAffinity: ClientIP
```

6.4 服务发现

微服务架构是一种新流行的架构模式，相比于传统的单块架构模式，微服务架构提倡将应用划分成一组小的服务。每个服务运行在其独立的进程中，服务之间互相协调、互相配合，服务与服务间采用轻量级的通信机制互相沟通。每个服务都围绕着具体业务进行构建，并且能够被独立部署和运行。

可以说 Docker 的轻量级容器技术天然适合微服务架构，每一个微服务都通过 Docker 进行打包、部署和运行。而 Docker 的集装箱能力为微服务架构保驾护航，可以快速、轻松地实现每个组件的测试和升级，将微服务架构的优势最大化。

但是应用的微服务化也将带来新的挑战，其中一个就是把应用划分成多个分布式组件运行，每个组件又将进行集群化扩展，组件和组件之间的相互发现和通信将会变得复杂起来，而一套服务编排机制就显得非常重要。

Kubernetes 提供了强大的服务编排能力，微服务化应用的每一个组件都以 Service 进行

抽象，组件和组件之间只需要访问 Service 即可以互相通信，而无须感知组件的集群变化。同时 Kubernetes 为 Service 提供了服务发现的能力，组件和组件之间可以简单地互相发现。

Kubernetes 中支持两种服务发现方法：环境变量和 DNS。

6.4.1 环境变量

当有 Pod 被创建的时候，Kubernetes 将为 Pod 设置每一个 Service 的相关环境变量，这些环境变量包括两种类型。

- **Kubernetes Service 环境变量**

  Kubernetes 为 Service 设置的环境变量形式，包括：

  ```
  {SVCNAME}_SERVICE_HOST
  {SVCNAME}_SERVICE_PORT
  {SVCNAME}_SERVICE_PORT_{PORTNAME}
  ```

  其中的服务名和端口名转为大写，连字符转换为下画线。

- **Docker Link 环境变量**

  相当于通过 Docker 的--link 参数实现容器连接时设置的环境变量形式，可参考 https://docs.docker.com/userguide/dockerlinks/。

比如现在有一个 Service，查询信息如下：

```
$ kubectl describe service my-dns
Name:                   my-dns
Namespace:              default
Labels:                 <none>
Selector:               run=my-dns
Type:                   ClusterIP
IP:                     10.254.105.183
Port:                   dns  53/UDP
Endpoints:              10.0.10.24:53
Port:                   dns-tcp  53/TCP
Endpoints:              10.0.10.24:53
Session Affinity:       None
No events.
```

这个 Service 对应的环境变量如下：

```
# Kubernetes Service 环境变量
MY_DNS_SERVICE_HOST=10.254.105.183
MY_DNS_SERVICE_PORT_DNS=53
MY_DNS_SERVICE_PORT_DNS_TCP=53
MY_DNS_SERVICE_PORT=53

# Docker Link 环境变量
MY_DNS_PORT_53_UDP_PORT=53
MY_DNS_PORT_53_TCP_ADDR=10.254.105.183
MY_DNS_PORT=udp://10.254.105.183:53
MY_DNS_PORT_53_UDP=udp://10.254.105.183:53
MY_DNS_PORT_53_UDP_PROTO=udp
MY_DNS_PORT_53_UDP_ADDR=10.254.105.183
MY_DNS_PORT_53_TCP_PORT=53
MY_DNS_PORT_53_TCP=tcp://10.254.105.183:53
MY_DNS_PORT_53_TCP_PROTO=tcp
```

可以看到，环境变量中记录了 Service 的虚拟 IP 以及端口和协议信息。这样一来，Pod 中的程序就可以使用这些环境变量发现 Service。

环境变量服务发现方式是 Kubernetes 默认支持的，但是此种方式存在限制。首先环境变量是租户隔离的，即 Pod 只能获取同 Namespace 中的 Service 的环境变量。另外，Pod 和 Service 的创建顺序是有要求的，即 Service 必须在 Pod 创建之前被创建，否则 Service 环境变量不会设置到 Pod 中。DNS 服务发现方式则没有这些限制。

6.4.2 DNS

DNS 服务发现方式需要 Kubernetes 提供 Cluster DNS 支持，Cluster DNS 会监控 Kubernetes API，为每一个 Service 创建 DNS 记录用于域名解析，这样在 Pod 中就可以通过 DNS 域名获取 Service 的访问地址。而对于一个 Service，Cluster DNS 会创建两条 DNS 记录：

```
[service_name].[namespace_name].[cluster_domain]
[service_name].[namespace_name].svc.[cluster_domain]
```

Cluster DNS 的安装方法可参考 2.3.1 节，本书使用的 Cluster DNS Server 的 IP 为 10.254.10.2，

Cluster DNS 的本地域为 cluster.local。

比如有一个 Service 的名称是 my-service，Namespace 是 my-ns，Cluster DNS 会创建 DNS 记录：

```
my-service.my-ns.cluster.local
my-service.my-ns.svc.cluster.local
```

对于如何通过 Cluster DNS 进行 Service 的域名解析，需要了解 Linux 的 DNS 域名解析机制。Linux 系统中的 DNS 域名解析是通过/etc/resolv.conf 进行配置的，它的格式很简单，每行以一个关键字开头，后接配置参数。/etc/resolv.conf 中的相关关键字说明如表 6-1 所示。

表 6-1 /etc/resolv.conf 关键字

关键字	说明
nameserver	DNS服务器的IP地址，可以有很多行的nameserver，每一个带一个IP地址，在查询时就按nameserver在本文件中的顺序进行
domain	主机的域名，当为没有域名的主机进行DNS查询时使用
search	DNS服务器搜索域，它的多个参数指明域名查询顺序。当要查询没有域名的主机时，主机将在由search声明的域中分别查找
options	DNS解析选项值，以Key/Value对的方式出现

Pod 中的容器使用容器宿主机的 DNS 域名解析配置，称为默认 DNS 配置。另外，如果 Kubernetes 部署并设置了 Cluster DNS 支持，那么在创建 Pod 的时候，默认会将 Cluster DNS 的配置写入 Pod 中容器的 DNS 域名解析配置中，称为 Cluster DNS 配置。

比如，Kubernetes Node 的/etc/resolv.conf 配置如下：

```
# Generated by NetworkManager
nameserver 218.85.157.99
```

在 Pod 的定义中通过.spec.dnsPolicy 设置 Pod 的 DNS 策略，默认值是 ClusterFirst，查看 DNS 策略设置为 ClusterFirst 的 Pod 中容器的/etc/resolv.conf：

```
$ kubectl exec my-app -- cat /etc/resolv.conf
nameserver 10.254.10.2
nameserver 218.85.157.99
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

其中 nameserver 10.254.10.2 是 Cluster DNS 配置，nameserver 218.85.157.99 是容器宿主

机的默认 DNS 配置。根据先后顺序，会优先使用 Cluster DNS 进行域名解析。另外，配置中包括根据 Pod 的 Namespace 和 Cluster Domain 设置 DNS 服务器搜索域：

```
search [namespace_name].svc.[cluster_domain] svc.[cluster_domain] [cluster_domain]
```

比如在 Namespace my-ns 下的 Pod 的 DNS 服务器的搜索域：

```
search my-ns.svc.cluster.local svc.cluster.local cluster.local
```

因为设置了 DNS 服务器的搜索域，在 Pod 中就可以使用[service_name].[namespace_name]访问到任意 Namespace 的 Service，而使用[service_name]可以访问到同 Namespace 下的 Service。比如对于 Namespace my-ns 下的 Service my-service，任意 Namespace 的 Pod 通过 my-service. my-ns 可以解析发现 Service：

```
$ kubectl exec my-pod -- nslookup my-service.my-ns --namespace=default
Server:  10.254.10.2
Address: 10.254.10.2#53

Name:    my-service.my-ns.svc.cluster.local
Address:  10.254.0.235
```

同 Namespace 下的 Pod 可以通过 my-service 解析发现 Service：

```
$ kubectl exec my-pod -- nslookup my-service --namespace=my-ns
Server:  10.254.10.2
Address: 10.254.10.2#53

Name:    my-service.my-ns.svc.cluster.local
Address: 10.254.0.235
```

另外，Cluster DNS 会为 Headless Service 的域名添加其 Endpoints 的所有 IP，即可以实现 DNS 的负载均衡：

```
$ kubectl describe service my-service
Name:              my-service
Namespace:         default
Labels:            <none>
Selector:          app=nginx
Type:              ClusterIP
IP:                None
Port:              <unnamed>80/TCP
Endpoints:         10.0.62.19:80, 10.0.62.20:80, 10.0.62.21:80
```

```
Session Affinity: None
No events.

$ kubectl exec my-pod -- nslookup my-service
Server:  10.254.10.2
Address: 10.254.10.2#53

Name:    my-service.default.svc.cluster.local
Address: 10.0.62.20
Name:    my-service.default.svc.cluster.local
Address: 10.0.62.19
Name:    my-service.default.svc.cluster.local
Address: 10.0.62.21
```

6.5 发布 Service

Service 的虚拟 IP 是由 Kubernetes 虚拟出来的内部网络，外部网络是无法寻址到的，但是有一些 Service 又需要对外暴露，比如 Web 前端。这时候就需要增加一层网络转发，即外网到内网的转发，Kubernetes 提供了 NodePort Service、LoadBalancer Service 和 Ingress 可以发布 Service。

6.5.1 NodePort Service

NodePort Service 是类型为 NodePort 的 Service，Kubernetes 除了会分配给 NodePort Service 一个内部的虚拟 IP，另外会在每一个 Node 上暴露端口 NodePort，外部网络可以通过[NodeIP]:[NodePort]访问到 Service。

我们现在创建一个 NodePort Service：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
```

```
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  type: NodePort
```

创建成功后查询 NodePort Service：

```
$ kubectl describe service my-nginx
Name:                   my-nginx
Namespace:              default
Labels:                 <none>
Selector:               app=nginx
Type:                   NodePort
IP:                     10.254.38.180
Port:                   http 80/TCP
NodePort:               http 32143/TCP
Endpoints:              <none>
Session Affinity:       None
No events.
```

可以看到，Kubernetes 给 NodePort Service 中每一个端口都创建了一个 NodePort（http 32143/TCP），在 NodePort Service 定义中可以通过.spec.ports[].nodePort 指定固定 NodePort，NodePort 的范围默认是 30000~32767，可以通过 Kubernetes API Server 的启动参数 --service-node-port-range 指定范围。

NodePort Service 就可以通过[NodeIP]:[NodePort]访问，而当 NodeIP 是一个公网 IP 时，外部就可以访问到 NodePort Service 了。

6.5.2 LoadBalancer Service

LoadBalancer Service 是类型为 LoadBalancer 的 Service，LoadBalancer Service 是建立在 NodePort Service 集群上的，Kubernetes 会分配给 LoadBalancer Service 一个内部的虚拟 IP，并且暴露 NodePort。除此之外，Kubernetes 请求底层云平台创建一个负载均衡器，将每个 Node 作为后端，负载均衡器将转发请求到[NodeIP]:[NodePort]。

LoadBalancer Service 需要底层云平台支持创建负载均衡器，比如 GCE，现在创建一个 LoadBalancer Service：

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  type: LoadBalancer
```

Kubernetes 会分配给 LoadBalancer Service 一个内部的虚拟 IP，并且暴露 NodePort。进一步的，Kubernetes 请求底层云平台创建一个负载均衡器，作为访问 LoadBalancer Service 的外部访问入口。负载均衡器由底层云平台创建提供，会包含一个 LoadBalancerIP，可以认为是 LoadBalancer Service 的外部 IP，查询 LoadBalancer Service：

```
$ kubectl get svc my-nginx
NAME       CLUSTER_IP       EXTERNAL_IP      PORT(S)   SELECTOR   AGE
my-nginx   10.254.147.57    78.110.25.19     80/TCP    app=nginx  4m
```

其中 EXTERNAL_IP:78.110.25.19 就是 LoadBalancer Service 的外部 IP。负载均衡器将每个 Node 作为后端，当请求 78.110.25.19:80 时，负载均衡器将转发请求到相应的 [NodeIP]:[NodePort]，从而访问到 LoadBalancer Service。

6.5.3 Ingress

Kubernetes 提供了一种 HTTP 方式的路由转发机制，称为 Ingress。Ingress 的实现需要两个组件支持，Ingress Controller 和 HTTP 代理服务器。HTTP 代理服务器将会转发外部的 HTTP 请求到 Service，而 Ingress Controller 则需要监控 Kubernetes API，实时更新 HTTP 代理服务器的转发规则。

提示

在当前版本（Kubernetes v1.1.1）中，Ingress 处于 beta 测试阶段。

我们现在创建一个 Ingress，Ingress 的定义文件 my-ingress.yaml：

```
apiVersion: extensions/v1beta1
```

```
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: my.example.com
    http:
      paths:
      - path: /app
        backend:
          serviceName: my-app
          servicePort: 80
```

Ingress 定义中的.spec.rules 设置了转发规则，其中配置了一条规则，当 HTTP 请求的 host 为 my.example.com 且 path 为/app 时，转发到 Service my-app 的 80 端口。

通过定义文件创建 Ingress：

```
$ kubectl create -f my-ingress.yaml
ingress "my-ingress" created
```

创建成功后可以查询 Ingress：

```
$ kubectl get ingress my-ingress
NAME         RULE          BACKEND   ADDRESS
my-ingress   -
             my.example.com
             /app          my-app:80
```

当 Ingress 创建成功后，需要 Ingress Controller 根据 Ingress 的配置，设置 HTTP 代理服务器的转发策略，外部通过 HTTP 代理服务器就可以访问到 Service。

在当前版本（Kubernetes v1.1.1）中，Ingress Controller 和 HTTP 代理服务器是作为外部组件运行的。官方提供了 GCE Load-Balancer 作为 HTTP 代理服务器，另外也可以使用 HAProxy 或者 Nginx 等开源方案。

以 Nginx 为例，将 Nginx 配置文件以模板形式编写：

```
const (
   nginxConf = `
events {
  worker_connections 1024;
```

```
}
http {
{{range $ing := .Items}}
{{range $rule := $ing.Spec.Rules}}
  server {
    listen 80;
    server_name {{$rule.Host}};
    resolver 127.0.0.1;
{{ range $path := $rule.HTTP.Paths }}
    location {{$path.Path}} {
      proxy_pass http://{{$path.Backend.ServiceName}}:{{$path.Backend.ServicePort}};
    }{{end}}
  }{{end}}{{end}}
}`
)
```

Ingress Controller 监控 Kubernetes API：

```
for {
    rateLimiter.Accept()
    ingresses, err := ingClient.List(labels.Everything(), fields.Everything())
    if err != nil || reflect.DeepEqual(ingresses.Items, known.Items) {
        continue
    }
    if w, err := os.Create("/etc/nginx/nginx.conf"); err != nil {
        log.Fatalf("Failed to open %v: %v", nginxConf, err)
    } else if err := tmpl.Execute(w, ingresses); err != nil {
        log.Fatalf("Failed to write template %v", err)
    }
    shellOut("nginx -s reload")
}
```

最终生成的 Nginx 配置文件如下：

```
events {
  worker_connections 1024;
}
http {
  server {
    listen 80;
    server_name my.example.com;
```

```
    resolver 127.0.0.1;

    location /app {
      proxy_pass http://my-app:80;
    }
  }
}
```

这样一来，Nginx 将作为访问入口，访问 http://my.example.com/app 的请求将会转发到 http://my-app:80，即访问到 Service。

第 7 章

数据卷

数据卷用于实现容器持久化数据，Kubernetes 对于数据卷重新定义，提供了丰富强大的功能。本章将数据卷按照功能划分为三类：本地数据卷、网络数据卷和信息数据卷，并一一进行说明，包括使用方法、配置参数和示例。另外，其中结合示例详细介绍了 Persistent Volume 和 Persistent Volume Claim。

7.1 Kubernetes 数据卷

在 Docker 的设计实现中，容器中的数据是临时的，即当容器被销毁时，其中的数据将会丢失。如果需要持久化数据，需要使用 Docker 数据卷挂载宿主机上的文件或者目录到容器中。

在 Kubernetes 系统中，当 Pod 重建的时候，数据是会丢失的，Kubernetes 也是通过数据卷来提供 Pod 数据的持久化的。Kubernetes 数据卷是对 Docker 数据卷的扩展，Kubernetes 数据卷是 Pod 级别的，可以用来实现 Pod 中容器的文件共享。

Kubernetes 数据卷适配对接各种存储系统，提供了丰富强大的功能。Kubernetes 提供了以下类型的数据卷：

- EmptyDir
- HostPath
- GCE Persistent Disk

- Aws Elastic Block Store
- NFS
- iSCSI
- Flocker
- GlusterFS
- RBD
- Git Repo
- Secret
- Persistent Volume Claim
- Downward API

7.2 本地数据卷

Kubernetes 中有两种类型的数据卷，它们只能作用于本地文件系统，我们称为本地数据卷。本地数据卷中的数据只会存在于一台机器上，所以当 Pod 发生迁移的时候，数据便会丢失，无法满足真正的数据持久化要求。但是本地数据卷提供了其他用途，比如 Pod 中容器的文件共享，或者共享宿主机的文件系统。

7.2.1 EmptyDir

EmptyDir 从名称上的意思看是空的目录，它是在 Pod 创建的时候新建的一个目录。

如果 Pod 配置了 EmptyDir 数据卷，在 Pod 的生命周期内都会存在，当 Pod 被分配到 Node 上的时候，会在 Node 上创建 EmptyDir 数据卷，并挂载到 Pod 的容器中。只要 Pod 存在，EmptyDir 数据卷都会存在（容器删除不会导致 EmptyDir 数据卷丢失数据），但是如果 Pod 的生命周期终结（Pod 被删除），EmptyDir 数据卷也会被删除，并且永久丢失。

EmptyDir 数据卷非常适合实现 Pod 中容器的文件共享。Pod 的设计提供了一个很好的容器组合的模型，容器之间各司其职，通过共享文件目录来完成交互，比如可以通过一个专职日志收集容器，在每个 Pod 中和业务容器中进行组合，来完成日志的收集和汇总。

我们创建一个 Pod，Pod 中包含两个容器，容器 synthetic-logger 写日志到/var/log 目录，而容器 sidecar-log-collector 负责收集/var/log 目录下的日志文件，然后导出到 Elasticsearch，其中的/var/log 目录就是一个 EmptyDir 数据卷，分别挂载到两个容器中，从而实现文件共

享。Pod 的定义文件如下：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    example: logging-sidecar
  name: logging-sidecar-example
spec:
  containers:
  - name: synthetic-logger
    image: ubuntu:14.04
    command: ["bash","-c","i=\"0\"; while true; do echo \"`hostname`: $i \" >>
/var/log/synthetic-count.log; date --rfc-3339 ns >> /var/log/synthetic-dates.log; sleep 4;
i=$[$i+1]; done"]
    volumeMounts:
    - name: log-storage
      mountPath: /var/log
  - name: sidecar-log-collector
    image: gcr.io/google_containers/fluentd-sidecar-es:1.2
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
    env:
    - name: FILES_TO_COLLECT
      value: "/var/log/synthetic-count.log /var/log/synthetic-dates.log"
    volumeMounts:
    - name: log-storage
      readOnly: true
      mountPath: /var/log
  volumes:
  - name: log-storage
    emptyDir: {}
```

7.2.2 HostPath

HostPath 数据卷允许将容器宿主机上的文件系统挂载到 Pod 中。如果 Pod 需要使用宿主机上的某些文件，可以使用 HostPath 数据卷。

创建一个 Pod 需要使用宿主机的 SSL 证书，可以通过创建 HostPath 数据卷，然后将宿主机的/etc/ssl/certs 目录挂载到容器中，Pod 的定义文件如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    volumeMounts:
      - name: ssl-certs
        mountPath: /etc/ssl/certs
        readOnly: true
  volumes:
    - name: ssl-certs
      hostPath:
        path: /etc/ssl/certs
```

7.3 网络数据卷

Kubernetes 提供了很多类型的数据卷以集成第三方的存储系统，包括一些非常流行的分布式文件系统，也有在 IaaS 平台上提供的存储支持，这些存储系统都是分布式的，通过网络共享文件系统，因此我们称这一类数据卷为网络数据卷。

网络数据卷能够满足数据的持久化需求，Pod 通过配置使用网络数据卷，每次 Pod 创建的时候都会将存储系统的远端文件目录挂载到容器中，数据卷中的数据将被永久保存，即使 Pod 被删除的时候，只是除去挂载数据卷，数据卷中的数据仍然保存在存储系统中，并且当新的 Pod 被创建的时候，仍是挂载同样的数据卷。

7.3.1 NFS

NFS（Network File System）即网络文件系统，是 FreeBSD 支持的一种文件系统，它允

许网络中的计算机通过 TCP/IP 共享资源。在 NFS 的应用中，本地 NFS 的客户端应用可以透明地读写位于远端 NFS 服务器上的文件，就像访问本地文件一样。

NFS 数据卷的配置参数如表 7-1 所示。

表 7-1　NFS 数据卷的配置参数

参数	可选项	说明
server	否	NFS的服务端地址
path	否	NFS的共享目录路径
readOnly	是	是否只读，默认为false（即可读可写）。

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
        - name: nfs
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: nfs
      nfs:
        server: nfs-server.default.kube.local
        path: "/"
```

7.3.2　iSCSI

iSCSI 技术是一种由 IBM 公司研究开发的，是一个供硬件设备使用的可以在 IP 的上层运行的 SCSI 指令集，这种指令集合可以实现在 IP 网络上运行 SCSI 协议，使其能够在诸如高速千兆以太网上进行路由选择。iSCSI 技术是一种新的储存技术，该技术是将现有 SCSI

接口与以太网络（Ethernet）技术结合，使服务器可与使用 IP 网络的储存装置互相交换资料。

iSCSI 数据卷的配置参数如表 7-2 所示。

表 7-2 iSCSI 数据卷配置参数

参数	可选项	说明
targetPortal	否	iSCSI Target服务地址
iqn	否	iSCSI的IQN号
lun	否	iSCSI的逻辑单元号
fsType	否	文件系统类型，ext4、xfs或ntfs
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: iscsipd
spec:
  containers:
  - image: kubernetes/pause
    name: iscsipd-ro
    volumeMounts:
    - mountPath: /mnt/iscsipd
      name: iscsipd-ro
  - image: kubernetes/pause
    name: iscsipd-rw
    volumeMounts:
    - mountPath: /mnt/iscsipd
      name: iscsipd-rw
  volumes:
  - iscsi:
      fsType: ext4
      iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
      lun: 0
      readOnly: true
      targetPortal: 10.0.2.15:3260
    name: iscsipd-ro
```

```
  - iscsi:
      fsType: ext4
      iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
      lun: 1
      targetPortal: 10.0.2.15:3260
    name: iscsipd-rw
```

7.3.3 GlusterFS

GlusterFS 是 Scale-Out 存储解决方案 Gluster 的核心，它是一个开源的分布式文件系统，具有强大的横向扩展能力，通过扩展能够支持数 PB 存储容量和处理数千客户端。GlusterFS 借助 TCP/IP 或 InfiniBand RDMA 网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。GlusterFS 基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能。

GlusterFS 数据卷配置参数如表 7-3 所示。

表 7-3 GlusterFS 数据卷配置参数

参数	可选项	说明
endpoints	否	GlusterFS服务端对应的Endpoint
path	否	GlusterFS数据卷路径
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Endpoints
metadata:
  name: glusterfs-cluster
subsets:
- addresses:
  - ip: 10.240.106.152
  ports:
  - port: 1
- addresses:
  - ip: 10.240.79.157
  ports:
  - port: 1
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: glusterfs
spec:
  containers:
  - image: kubernetes/pause
    name: glusterfs
    volumeMounts:
    - mountPath: /mnt/glusterfs
      name: glusterfsvol
  volumes:
  - glusterfs:
      endpoints: glusterfs-cluster
      path: kube_vol
      readOnly: true
    name: glusterfsvol
```

7.3.4 RBD（Ceph Block Device）

Ceph 是开源、分布式的网络存储，同时又是文件系统。Ceph 的设计目标是卓越的性能、可靠性以及可扩展性。Ceph 基于可靠的、可扩展的和分布式的对象存储，通过一个分布式的集群管理元数据，符合 POSIX。RBD（Rados Block Device）是一个 Linux 块设备驱动，提供了一个共享网络块设备，实现与 Ceph 的交互。RBD 在 Ceph 对象存储的集群上进行条带化和复制，提供可靠性、可扩展性以及对块设备的访问。

Kubernetes 中支持 RBD 方式，RBD 数据卷配置的参数如表 7-4 所示。

表 7-4 RBD 数据卷配置参数

参数	可选项	说明
monitors	否	Ceph Monitors的地址
image	否	Rados 镜像名称
fsType	否	文件系统类型，ext4、xfs或ntfs
pool	是	Rados Pool名称，默认是rbd

续表

参数	可选项	说明
user	是	Rados用户名，默认是admin
keyring	是	Rados钥匙圈文件的路径，默认为/etc/ceph/keyring
secretRef	是	Rados用户认证Secret，默认为空
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: rbd
spec:
  containers:
  - image: kubernetes/pause
    name: rbd-rw
    volumeMounts:
    - mountPath: /mnt/rbd
      name: rbdpd
  volumes:
  - name: rbdpd
    rbd:
      fsType: ext4
      image: foo
      keyring: /etc/ceph/keyring
      monitors:
      - 10.16.154.78:6789
      - 10.16.154.82:6789
      - 10.16.154.83:6789
      pool: kube
      readOnly: true
      user: admin
```

7.3.5 Flocker

Flocker 是一个容器数据管理工具，作为 ClusterHQ 公司 2014 年推出的产品，Flocker 主要负责 Docker 容器及其数据的管理。从功能方面而言，Flocker 是一个数据卷管理器和

多主机的 Docker 集群管理工具。用户可以通过它来控制数据，实现在 Docker 中运行数据库、队列和键值（Key/Value）存储等服务，并在应用程序中轻松使用这些服务。

Flocker 数据卷配置参数如表 7-5 所示。

表 7-5 Flocker 数据卷配置参数

参数	可选项	说明
datasetName	否	Flocker数据集名称

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: flocker-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
          # name must match the volume name below
          - name: www-root
            mountPath: "/usr/share/nginx/html"
  volumes:
    - name: www-root
      flocker:
        datasetName: my-flocker-vol
```

7.3.6 AWS Elastic Block Store

Amazon Elastic Block Store（Amazon EBS）在 AWS 云中提供用于 Amazon EC2 实例的持久性数据的块级存储卷。如果 Kubernetes 运行在 AWS 之上，就可以非常方便地使用 Amazon Elastic Block Store 作为数据卷。

Amazon Elastic Block Store 数据卷的配置参数如表 7-6 所示。

表 7-6　Amazon Elastic Block Store 数据卷配置参数

参数	可选项	说明
volumeID	否	EBS数据卷标识
fsType	否	文件系统类型，ext4、xfs或ntfs
partition	是	磁盘挂载分区
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: aws://<availability-zone>/<volume-id>
      fsType: ext4
```

7.3.7　GCE Persistent Disk

GCE Persistent Disk 是 GCE 提供的持久化数据的服务，如果 Kubernetes 运行在 GCE 之上，可以使用 GCE Persistent Disk 作为数据卷。

GCE Persistent Disk 数据卷的配置参数如表 7-7 所示。

表 7-7　GCE Persistent Disk 数据卷配置参数

参数	可选项	说明
pdName	否	GCE Persistent Disk的名称

续表

参数	可选项	说明
fsType	否	文件系统类型，ext4、xfs或ntfs
partition	是	磁盘挂载分区
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

7.4 Persistent Volume 和 Persistent Volume Claim

理解每个存储系统是一件复杂的事情，特别是对于普通用户来说，有时候并不关心各种存储实现，只希望能够安全可靠地存储数据。Kubernetes 中提供了 Persistent Volume 和 Persistent Volume Claim 机制，这是存储消费模式。Persistent Volume 是由系统管理员配置创建的一个数据卷，它代表了某一类存储插件实现，可以是 NFS、iSCSI 等；而对于普通用户来说，通过 Persistent Volume Claim 可请求并获得合适的 Persistent Volume，而无须感知后端的存储实现。

Persistent Volume Claim 和 Persistent Volume 的关系其实类似于 Pod 和 Node，Pod 消费

Node 的资源，Persistent Volume Claim 则是消费 Persistent Volume 的资源。Persistent Volume 和 Persistent Volume Claim 相互关联，有着完整的生命周期管理。

- **准备**
  系统管理员规划并创建一系列的 Persistent Volume，Persistent Volume 在创建成功后处于可用状态。
- **绑定**
  用户创建 Persistent Volume Claim 来声明存储请求，包括存储大小和访问模式。Persistent Volume Claim 创建成功后进入等待状态，当 Kubernetes 发现有新的 Persistent Volume Claim 创建的时候，就会根据条件查找 Persistent Volume。当有 Persistent Volume 匹配的时候，就会将 Persistent Volume Claim 和 Persistent Volume 进行绑定，Persistent Volume 和 Persistent Volume Claim 都进入绑定状态。
  Kubernetes 只会选择可用状态的 Persistent Volume，并且采取最小满足策略，而当没有 Persistent Volume 满足需求的时候，Persistent Volume Claim 将处于等待阶段。比如现在有两个 Persistent Volume 可用，一个 Persistent Volume 的容量是 50Gi，一个 Persistent Volume 的容量是 60Gi。那么请求 40Gi 的 Persistent Volume Claim 会被绑定到 50Gi 的 Persistent Volume，而请求 100Gi 的 Persistent Volume Claim 则处于等待状态，直到有大于 100Gi 的 Persistent Volume 出现（Persistent Volume 有可能被新建或者被回收）。
- **使用**
  创建 Pod 的时候使用 Persistent Volume Claim，Kubernetes 便会查询其绑定的 Persistent Volume，去调用真正的存储实现，然后将数据卷挂载到 Pod 中。
- **释放**
  当用户删除 Persistent Volume 所绑定的 Persistent Volume Claim 时，Persistent Volume 则进入释放状态。此时 Persistent Volume 中可能残留着之前 Persistent Volume Claim 使用的数据，所以 Persistent Volume 并不可用，需要对 Persistent Volume 进行回收操作。
- **回收**
  释放的 Persistent Volume 需要回收才能再次使用，回收的策略可以是人工处理，或者由 Kubernetes 自动进行清理，如清理失败，Persistent Volume 会进入失败状态。

综上所述，Persistent Volume 的状态包括如下几项。

- Available：Persistent Volume 创建成功后进入可用状态，等待 Persistent Volume Claim 消费。
- Bound：Persistent Volume 被分配到 Persistent Volume Claim 进行绑定，Persistent Volume 进入绑定状态。
- Released：Persistent Volume 绑定的 Persistent Volume Claim 被删除，Persistent Volume 进入释放状态，等待回收处理。
- Failed：Persistent Volume 执行自动清理回收策略失败后，Persistent Volume 会进入失败状态。

Persistent Volume Claim 的状态包括如下几项。

- Pending：Persistent Volume Claim 创建成功后进入等待状态，等待绑定 Persistent Volume。
- Bound：分配 Persistent Volume 给 Persistent Volume Claim 进行绑定，Persistent Volume Claim 进入绑定状态。

7.4.1 创建 Persistent Volume

Persistent Volume 代表可靠的存储系统，从实现上来说，Persistent Volume 实际上就是复用了已有的数据卷实现，配置方法也是一致的，Persistent Volume 目前支持以下类型：

- GCE Persistent Disk
- AWS Elastic Block Store
- NFS
- iSCSI
- RBD（Ceph Block Device）
- GlusterFS
- HostPath：只适合测试使用

Persistent Volume 是需要事先创建好的，这一般来说是系统管理员的工作。系统管理员根据实际情况，创建一系列可用的 Persistent Volume。比如 Kubernetes 是运行在 AWS 之上的，购买了 10 个 100Gi 的 Amazon EBS，那就可以创建 10 个容量为 100Gi 的 Persistent Volume。

创建 Persistent Volume 的时候需要指定数据卷的容量、访问模式和回收策略。

- **容量**

  Persistent Volume 通过设置资源容量，然后 Persistent Volume Claim 在请求的时候指定资源需求来进行匹配。比如现在有 1 个容量为 10Gi 的 Persistent Volume 和 1 个容量为 20Gi 的 Persistent Volume，Persistent Volume Claim 请求 15Gi，就匹配了 20Gi 的 Persistent Volume。目前 Persistent Volume 的资源容量只有一个属性，就是存储大小：

```
capacity:
  storage: 20Gi
```

- **访问模式**
  - ReadWriteOnce：数据卷能够在一个节点上挂载为读写目录。
  - ReadOnlyMany：数据卷能够在多个节点上挂载为只读目录。
  - ReadWriteMany：数据卷能够在多个节点上挂载为读写目录。
- **回收策略**

  当前支持的回收策略如下所示。
  - Retain：Persistent Volume 在释放后，需要人工进行回收操作。
  - Recycle：Persistent Volume 在释放后，Kubernetes 自动进行清理，清理成功后 Persistent Volume 则可以再次绑定使用。目前只有 NFS 和 HostPath 类型的 Persistent Volume 支持回收策略。当执行回收策略的时候，会创建一个 Persistent Volume Recycler Pod，这个 Pod 执行清理动作，即删除 Persistent Volume 目录下的所有文件（包括隐藏文件）。

现在创建一个 Persistent Volume，Persistent Volume 的定义文件 nfs-pv.yaml：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
  labels:
    type: nfs
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs-server
    path: /
```

Persistent Volume 的定义中使用了 NFS，配置参数同 NFS 数据卷一致，另外设置了存储容量为 5Gi，访问权限为 ReadWriteMany。

通过定义文件创建 Persistent Volume：

```
$ kubectl create -f nfs-pv.yaml
persistentvolume "nfs-pv" created
```

创建成功后可以查询创建的 Persistent Volume，Persistent Volume 的状态为 Available：

```
$ kubectl describe persistentvolume nfs-pv
Name:          nfs-pv
Labels:        type=nfs
Status:        Available
Claim:
Reclaim Policy:    Retain
Access Modes: RWX
Capacity:          5Gi
Message:
Source:
    Type: NFS (an NFS mount that lasts the lifetime of a pod)
    Server:nfs-server
    Path: /
    ReadOnly:false
```

7.4.2 创建 Persistent Volume Claim

Persistent Volume Claim 指定所需要的存储大小，然后 Kubernetes 会选择满足条件的 Persistent Volume 进行绑定。

现在创建 Persistent Volume Claim 来消费刚创建的 Persistent Volume，Persistent Volume Claim 的定义文件 test-pvc.yaml：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
```

```
    requests:
      storage: 3Gi
```

通过定义文件创建 Persistent Volume Claim：

```
$ kubectl create -f my-pvc.yaml
persistentvolumeclaim "my-pvc" created
```

创建成功后可以查询创建的 Persistent Volume Claim：

```
$ kubectl describe persistentvolumeclaim my-pvc
Name:         my-pvc
Namespace:    default
Status:       Bound
Volume:       nfs-pv
Labels:       <none>
Capacity:     5Gi
Access Modes: RWX
```

可以看到，Persistent Volume Claim 已经绑定 Persistent Volume 为 nfs-pv，然后查询 Persistent Volume 的状态为 Bound：

```
$ kubectl describe persistentvolume nfs-pv
Name:              nfs-pv
Labels:            type=nfs
Status:            Bound
Claim:             default/my-pvc
Reclaim Policy:    Retain
Access Modes:      RWX
Capacity:          5Gi
Message:
Source:
    Type: NFS (an NFS mount that lasts the lifetime of a pod)
    Server:nfs-server
    Path: /
    ReadOnly:false
```

最后创建 Pod 来使用 Persistent Volume Claim，Pod 的定义文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
```

```
    labels:
      app: busybox
  spec:
    containers:
    - name: busybox
      image: busybox
      command:
      - sleep
      - 3600
      volumeMounts:
      - mountPath: /busybox-data
        name: data
    volumes:
    - name: data
     persistentVolumeClaim:
       claimName: my-pvc
```

7.5 信息数据卷

Kubernetes 中有一些数据卷，主要用来给容器传递配置信息，我们称之为信息数据卷，比如 Secret 和 Downward API，都是将 Pod 的信息以文件形式保存，然后以数据卷方式挂载到容器中，容器通过读取文件获取相应的信息。从功能设计上来说，是有点偏离数据卷的本意，数据卷是用来持久化数据的，或者进行文件共享的。未来版本可能会对这部分进行重构，将信息数据卷提供的功能放在更合适的地方。

7.5.1 Secret

Kubernetes 提供了 Secret 来处理敏感数据，比如密码、Token 和密钥，相比于直接将敏感数据配置在 Pod 的定义或者镜像中，Secret 提供了更加安全的机制，防止数据泄露。

Secret 的创建是独立于 Pod 的，以数据卷的形式挂载到 Pod 中，Secret 的数据将以文件的形式保存，容器通过读取文件可以获取需要的数据。

Secret 的类型有 3 种。

- Opaque：自定义数据内容，默认值。

- kubernetes.io/service-account-token：Service Account 的认证内容，可参考 10.3 节。
- kubernetes.io/dockercfg：Docker 镜像仓库的认证内容，可参考 4.3.1 节。

现在有一个应用需要获取一个账号密码，即可以通过 Secret 来实现：

```
username: my-username
password: my-password
```

因为是自定义数据内容，所以 Secret 的类型是 Opaque，配置的数据是一系列 Key/Value 对，其中 Value 需要使用 Base64 加密，Secret 定义文件 secret.yaml：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: bXktdXNlcm5hbWUK
  password: bXktcGFzc3dvcmQK
```

通过定义文件创建 Secret：

```
$ kubectl create -f secret.yaml
secret "mysecret" created
```

创建成功后查询 Secret：

```
$ kubectl describe secret mysecret
Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations:  <none>
Type:         Opaque

Data
====
password:     12 bytes
username:     12 bytes
```

然后创建 Pod 使用该 Secret，Pod 的定义文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    app: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command:
    - sleep
    - "3600"
    volumeMounts:
    - mountPath: /secret
      name: secret
      readOnly: true
  volumes:
  - name: secret
    secret:
     secretName: mysecret
```

Pod 的定义中声明了 Secret 数据卷，并且将 Secret 数据卷挂载到了 Pod 容器中的/secret 目录下，因为 Secret 包含两个 Key/Value 对，在容器的/secret 目录中分别有两个文件 username 和 password，文件的内容就是解密后的数据：

```
$ kubectl exec busybox -- ls /secret
password
username

$ kubectl exec busybox -- cat /secret/username
my-username

$ kubectl exec busybox -- cat /secret/password
my-password
```

7.5.2 Downward API

Downward API 可以通过环境变量的方式告诉容器 Pod 的信息（可参考 4.3.3 节），另外，

也可以通过数据卷方式传值，Pod 的信息将会以文件的形式通过数据卷挂载到容器中，在容器中可以通过读取文件获取信息，目前支持：

- Pod 的名称。
- Pod 的 Namespace。
- Pod 的 Label。
- Pod 的 Annotation。

创建 Pod 使用 Downward API 数据卷，Pod 的定义文件 downwardapi-volume.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-volume
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: ubuntu:14.04
      command: ["/bin/bash", "-c", "while true; do sleep 5; done"]
      volumeMounts:
        - name: podinfo
          mountPath: /podinfo/
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "pod_name"
            fieldRef:
              fieldPath: metadata.name
          - path: "pod_namespace"
            fieldRef:
```

```
      fieldPath: metadata.namespace
  - path: "pod_labels"
    fieldRef:
      fieldPath: metadata.labels
  - path: "pod_annotations"
    fieldRef:
      fieldPath: metadata.annotations
```

Pod 定义中声明了 Downward API 数据卷，分别引用了 Pod 的相关属性，然后将数据卷挂载到容器/podinfo 目录下。在 Pod 创建运行后，就可以查询/podinfo 目录下的数据：

```
$ kubectl exec downwardapi-volume -- ls /podinfo
pod_annotations
pod_labels
pod_name
pod_namespace

$ kubectl exec downwardapi-volume -- cat /podinfo/pod_name
downwardapi-volume

$ kubectl exec downwardapi-volume -- cat /podinfo/pod_namespace
default

$ kubectl exec downwardapi-volume -- cat /podinfo/pod_labels
cluster="test-cluster1"
rack="rack-22"

$ kubectl exec downwardapi-volume -- cat /podinfo/pod_annotations
build="two"
builder="john-doe"
kubectl.kubernetes.io/last-applied-configuration="..."
kubernetes.io/config.seen="2015-11-02T18:29:57.509960318+08:00"
kubernetes.io/config.source="api"
```

7.5.3 Git Repo

Kubernetes 支持将 Git 仓库下载到 Pod 中，目前是通过 Git Repo 数据卷实现，即当 Pod 配置 Git Repo 数据卷时，就下载配置的 Git 仓库到 Pod 的数据卷中，然后挂载到容器中。

我们现在定义一个 Pod 使用 Git Repo 数据卷，Pod 的定义文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    app: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command:
    - sleep
    - "3600"
    volumeMounts:
    - mountPath: /config
      name: busybox-config
  volumes:
  - name: busybox-config
    gitRepo:
      repository: https://github.com/wulonghui/busybox-config.git
      revision: master
```

Pod 的定义中声明了 Git Repo 数据卷，然后挂载到容器的/config 目录下，在 Pod 运行后，即会下载指定的 Git 仓库到/config 目录下：

```
$ kubectl exec busybox -- ls /config
busybox-config
```

第 8 章

访问 Kubernetes API

Kubernetes 规范和健全的 API 模型，为 Kubernetes 的使用和扩展提供了非常好的支持。

本章首先说明 Kubernetes 提供的 API 对象以及元数据，然后介绍 Kubernetes API 的访问方式，最后详细阐述 Kubernetes 的命令行工具，帮助读者真正掌握如何使用 Kubernetes。

8.1 API 对象与元数据

Kubernetes 中的很多功能是通过 API 对象来实现的，在前面的章节中我们已经创建过许多 API 对象，包括 Pod、Replication Controller、Service 和 Secret 等。在定义 API 对象的时候需要分别声明 API 版本（apiVersion）和类型（kind），当前版本 Kubernetes v1.1.1 支持的 API 对象的 API 版本和类型如下所示：

- v1/Pod
- v1/ReplicationController
- v1/Service
- v1/Endpoints
- v1/Events
- v1/Node
- v1/Namespace
- v1/Secret

- v1/ServiceAccount
- v1/PersistentVolume
- v1/PersistentVolumeClaim
- v1/LimitRange
- v1/ResourceQuota
- extensions/v1beta1/Deployment
- extensions/v1beta1/HorizontalPodAutoscaler
- extensions/v1beta1/Ingress
- extensions/v1beta1/Job
- extensions/v1beta1/Daemonset

API 对象的元数据用来定义 API 对象的基本信息，体现在定义中的 metadata 字段，包含以下属性。

- namespace：指定 API 对象所在的 Namespace。
- name：指定 API 对象的名称。
- labels：设置 API 对象的 Label。
- annotations：设置 API 对象的 Annotation。

Namespace

Namespace 是 Kubernetes 提供的多租户，不同的项目、团队或者用户可以通过 Namespace 进行区分管理，并且设置安全控制和其他策略。绝大部分 API 对象（除了 Node）归属于 Namespace，API 对象通过.metadata.namespace 指定 Namespace，如果没有指定 Namespace，那么就是归属于默认 Namespace default。

Name

名称是一个重要的属性，是人类可读的，元数据中的.metadata.name 用于指定 API 对象的名称。Kubernetes 系统中的 API 对象必须能够通过名称唯一标识，Kubernetes 包含 Namespace 的逻辑层级，大部分 API 对象必须归属于 Namespace，所以这些 API 对象的名称必须在 Namespace 内唯一。而另外对于 Node 和 Namespace 来说，需要在 Kubernetes 系统中唯一。

Label

Label 用于区分 API 对象的 Key/Value 对，Label 存放的应该是具有标识性的数据，

Kubernetes 通过 Label 可以对 API 对象进行选择。Replication Controller 和 Service 都是通过 Label 关联 Pod，而 Pod 也可以通过 Label 选择 Node。

Annotation

Annotation 用于存放用户的自定义数据，Annotation 存放的是非标识的数据，所以不能像 Label 一样进行对象选择。但是 Annotation 的数据可以是长数据，可以有结构或者无结构，作为 Label 的一种补充，Annotation 也是 Key/Value 对：

```
annotations:
  key1: value1
  key2: value2
```

8.2 如何访问 Kubernetes API

使用 Kubernetes 都需要访问其 API，而 Kubernetes API Server 作为 Kubernetes 系统的入口，以 REST API 接口方式提供给外部调用，所以访问 Kubernetes API 实际上就是调用 Kubernetes API Server。其中 Kubernetes API Server 集成了 Swagger，可以通过界面查询所有 API 的详细信息（http://kube-master:8080/swagger-ui/），如图 8-1 所示。

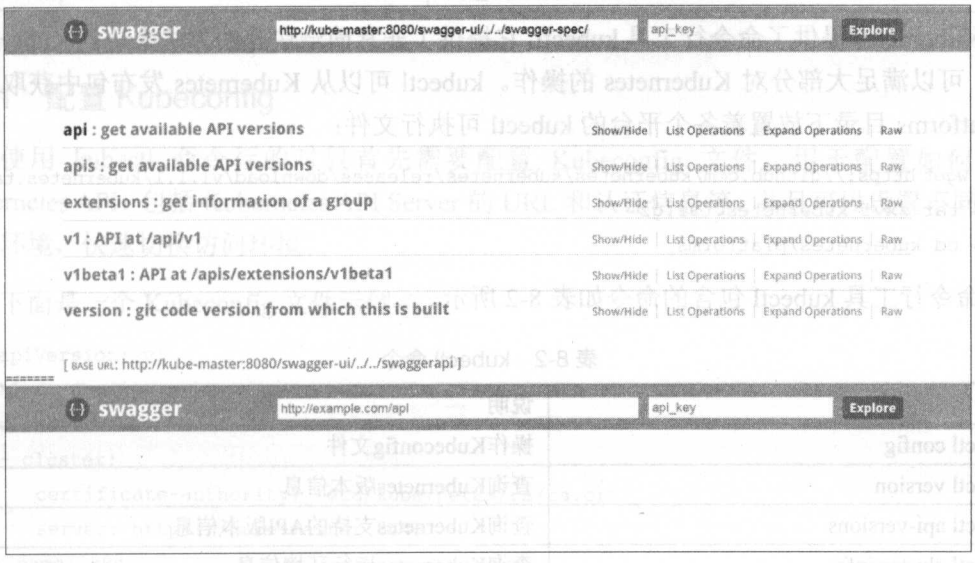


图 8-1 Kubernetes 集成 Swagger 提供 API 查询

除此之外，社区中封装提供了各种开发语言的 Kubernetes 客户端 Lib 包，如图 8-1 所示，可以使用这些 Lib 包来开发程序以访问 Kubernetes API。

表 8-1 Kubernetes 客户端 Lib 包

Lib包	语言	URL
Kubernetes Client	Go	https://github.com/kubernetes/kubernetes/tree/v1.1.1/pkg/client
Amdatu Kubernetes	Java	https://bitbucket.org/amdatulabs/amdatu-kubernetes
kubernetes-client	Java	https://github.com/fabric8io/kubernetes-client
kubr	Ruby	https://github.com/Ch00k/kubr
kubeclient	Ruby	https://github.com/abonas/kubeclient
kubernetes-client	PHP	https://github.com/maclof/kubernetes-client
kubernetes-api-php-client	PHP	https://github.com/devstub/kubernetes-api-php-client
node-kubernetes-client	Node.js	https://github.com/tenxcloud/node-kubernetes-client
Net::Kubernetes	Perl	https://github.com/kubernetes/kubernetes/blob/v1.1.1/docs/devel/client-libraries.md

8.3 使用命令行工具 kubectl

Kubernetes 提供了命令行工具 kubectl，它提供了非常简洁快速的方法来访问 Kubernetes API，可以满足大部分对 Kubernetes 的操作。kubectl 可以从 Kubernetes 发布包中获取，其中 platforms 目录下放置着各个平台的 kubectl 可执行文件：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/platforms
```

命令行工具 kubectl 包含的命令如表 8-2 所示。

表 8-2 kubectl 命令

命令	说明
kubectl config	操作Kubeconfig文件
kubectl version	查询Kubernetes版本信息
kubectl api-versions	查询Kubernetes支持的API版本信息
kubectl cluster-info	查询Kubernetes运行环境信息
kubectl proxy	为Kubernetes API Server启动服务代理

续表

命令	说明
kubectl create	创建API对象
kubectl delete	删除API对象
kubectl edit	编辑API对象
kubectl apply	更新API对象
kubectl patch	为API对象打补丁
kubectl label	操作API对象的Label
kubectl annotate	操作API对象的Annotation
kubectl logs	获取Pod中容器的输出日志
kubectl autoscale	执行Pod的自动伸缩
kubectl rolling-update	执行Pod的滚动升级
kubectl scale	执行Pod的弹性伸缩
kubectl attach	连接Pod中启动的容器
kubectl exec	在Pod的容器中执行命令
kubectl port-forward	为Pod设置端口转发
kubectl run	创建Replication Controller
kubectl expose	创建Service

8.3.1 配置 Kubeconfig

使用 kubectl 命令行的时候首先需要配置 Kubeconfig 文件，用于配置如何访问 Kubernetes API，包括 Kubernetes API Server 的 URL 和认证信息等，并且可以设置不同的上下文环境，快速切换访问环境。

下面是一个 Kubeconfig 文件示例：

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/ca/ca.crt
    server: https://kube-master:6443
  name: k8s
users:
```

```
- name: k8s-client
  user:
    client-certificate: /etc/kubernetes/ca/client.crt
    client-key: /etc/kubernetes/ca/client.key.insecure
- name: k8s-admin
  user:
    password: test
    username: k8s-admin
contexts:
- context:
    cluster: k8s
    user: k8s-admin
    namespace: default
  name: default
current-context: default
preferences: {}
```

Kubeconfig 文件的定义中包含以下关键部分。

- clusters：设置 Kubernetes API Server 的访问 URL 和相关属性。
- users：设置访问 Kubernetes API Server 的认证信息。
- contexts：设置 kubelet 执行上下文。
- current-context：设置 kubelet 执行当前上下文。
- preferences：设置 kubelet 其他属性。

Kubeconfig 文件可以手动进行编辑，也可以通过 kubectl config 命令进行查询和设置，如表 8-3 所示。

表 8-3 kubectl config 命令

命令	说明
kubectl config view	查看Kubeconfig文件
kubectl config set-cluster	设置Kubeconfig的clusters
kubectl config set-credentials	设置Kubeconfig的users
kubectl config set-context	设置Kubeconfig的contexts
kubectl config use-context	设置Kubeconfig的current-context

8.3.2 Kubernetes 操作

kubectl version

kubectl version 命令用来查询 Kubernetes 的版本信息，包括客户端和服务端，这在问题定位的时候特别有帮助：

```
$ kubectl version
Client Version: version.Info{Major:"1",Minor:"1",GitVersion:"v1.1.1",GitCommit:"...",
GitTreeState:"clean"}
Server Version: version.Info{Major:"1",Minor:"1",GitVersion:"v1.1.1",GitCommit:"...",
GitTreeState:"clean"}
```

kubectl api-versions

kubectl api-versions 命令可以查询 Kubernetes 支持的 API 版本：

```
$ kubectl  api-versions
extensions/v1beta1
v1
```

kubectl cluster-info

kubectl cluster-info 命令可以查询 Kubernetes 的运行环境信息，包括 Kubernetes API Server 和平台级别 Service（kubernetes.io/cluster-service=true）的地址：

```
$ kubectl cluster-info
Kubernetes master is running at http://k8s-master:8080
KubeDNS is running at http://k8s-master:8080/api/v1/proxy/namespaces/kube-system/services/
kube-dns
KubeUI is running at http://k8s-master:8080/api/v1/proxy/namespaces/kube-system/services/
kube-ui
...
```

kubectl proxy

kubectl proxy 命令可以为 Kubernetes API Server 在本地启动一个代理服务，访问这个代理服务就可以访问 Kubernetes API Server。

执行 kubectl proxy 的时候可以指定监听端口和 API 访问前缀：

```
$ kubectl proxy --port=8011 --api-prefix=/k8s-api
Starting to serve on 127.0.0.1:8011
```

kubectl proxy 启动后就可以通过 http://127.0.0.1:8011/k8s-api 访问到代理服务，从而访问 Kubernetes API Server。

8.3.3 API 对象操作

kubectl 命令行可以操作 API 对象，执行的时候需要指定 API 对象的类型，类型可以用单数形式、复数形式或者简写形式，具体如下所示：

- pod/pods/po
- replicationcontroller/replicationcontrollers/rc
- daemonset/daemonsets/ds
- service/services/svc
- endpoints/ep
- event/events/ev
- node/nodes/no
- namespace/namespaces/ns
- secret/secrets
- serviceaccount/serviceaccounts
- persistentvolume/persistentvolumes/pv
- persistentvolumeclaim/persistentvolumeclaims/pvc
- limitrange/limitranges/limits
- resourcequota/resourcequotas/quota
- componentstatuses/cs
- daemonset/daemonsets/ds
- deployment/deployments
- horizontalpodautoscaler/horizontalpodautoscalers/hpa
- ingress/ingresses/ing
- job/jobs

kubectl create

kubectl create 命令用来创建 API 对象，格式支持 JSON 和 YAML。使用 kubectl create 主要是通过定义文件进行创建：

```
$ kubectl create -f /path/to/file
```

也可通过传递标准输入（stdin）进行创建：

```
$ cat /path/to/file | kubectl create -f -
```

kubectl create 支持连续创建多个 API 对象：

```
$ kubectl create -f /path/to/file1 -f /path/to/file2
```

kubectl get

kubectl get 命令可以用来查询各类 API 对象的信息。

kubectl get 可以查询指定 API 对象：

```
$ kubectl get TYPE NAME
```

可以查询一个类型的所有 API 对象：

```
$ kubectl get TYPE
```

也可以同时查询多个类型，类型之间用,分隔：

```
$ kubectl get TYPE1,TYPE2
```

kubectl get 支持通过 Label 筛选 API 对象：

```
$ kubectl get TYPE --selector key1=value1,key2=value
```

默认情况下，kubectl get 只会显示简要信息，以下方式可以显示详细信息：

```
$ kubectl get TYPE NAME --output json
$ kubectl get TYPE NAME --output yaml
```

kubectl get 也支持通过 Go Template 或者 JSON Path 提取指定信息：

```
$ kubectl get TYPE NAME --output go-template=...
$ kubectl get TYPE NAME --output jsonpath=...
```

kubectl describe

kubectl describe 命令可以用来查询各类 API 对象的概况信息，支持批量查询和指定查询：

```
$ kubectl describe TYPE
$ kubectl describe TYPE NAME
```

kubectl delete

kubectl delete 命令用来删除 API 对象，删除的时候可以指定 API 对象进行删除：

```
$ kubectl delete TYPE NAME
```

也可以通过定义文件删除：

```
$ kubectl delete -f /path/to/file
```

另外，kubectl delete 命令可以进行批量删除，多个 API 对象类型之间用,分隔：

```
$ kubectl delete TYPE1,TYPE2 --all
```

或者通过 Label 筛选删除：

```
$ kubectl delete TYPE1,TYPE2 --selector key1=value1,key2=value
```

kubectl apply

kubectl apply 命令可以用来更新已创建的 API 对象，主要是通过定义文件进行修改：

```
$ kubectl apply -f /path/to/file
```

也可通过传递标准输入（stdin）进行修改：

```
$ cat /path/to/file | kubectl apply -f -
```

kubectl replace

kubectl replace 命令与 kubectl apply 类似，都可以用来更新已创建的 API 对象：

```
$ kubectl apply -f /path/to/file
```

但有时候 API 对象的有些属性无法直接更新，这时候可以使用 kubectl replace 命令强制进行重建以实现更新：

```
$ kubectl apply -f /path/to/file --force
```

kubectl edit

kubectl edit 命令可以用来编辑已创建的 API 对象，使用 kubectl edit 命令的时候会开启一个编辑器。通过编辑器可以方便地对 API 对象进行编辑，默认的编辑器是 VI，可以通过环境变量 KUBE_EDITOR 选择其他编辑器：

```
$ KUBE_EDITOR="nao" kubectl edit TYPE NAME
```

在编辑器中默认显示的格式是 YAML，也可以指定为 JSON：

```
$ kubectl edit TYPE NAME --output json
```

kubectl patch

kubectl patch 命令可以给 API 对象打补丁，即修改指定属性，这样可以非常方便地修改 API 对象，其中 PATCH 要使用 JSON 格式：

```
$ kubectl patch TYPE --patch PATCH
```

kubectl label

kubectl label 命令可以用来操作 API 对象的 Label，包括增加、修改和删除。

给 API 对象增加 Label，多个 Label 之间用空格分隔：

```
$ kubectl label TYPE NAME label1=value1 label2=value2
```

更新 API 对象已有的 Label，需要加上--overwrite 参数进行覆盖：

```
$ kubectl label TYPE NAME label1=new-value --overwrite
```

删除 API 对象已有的 Label，在 Label 的 KEY 后面加上-：

```
$ kubectl label TYPE NAME label1-
```

另外，kubectl label 也支持通过参数--all 和--selector 进行批量操作。

kubectl annotate

kubectl annotate 命令可以用来操作 API 对象的 Annotation，kubectl annotate 同 kubectl label 命令差不多，只不过操作的对象换成 Annotation。

给 API 对象增加 Annotation：

```
$ kubectl annotate TYPE NAME annotation1=value1 annotation2=value2
```

更新 API 对象已有的 Annotation，需要加上--overwrite 参数进行覆盖：

```
$ kubectl annotate TYPE NAME annotation1=new-value --overwrite
```

删除 API 对象已有的 Annotation，在 Annotation 的 KEY 后面加上-：

```
$ kubectl annotate TYPE NAME annotation1-
```

8.3.4 Pod 操作

kubectl logs

kubectl logs 命令用于打印 Pod 中容器的日志输出，如果 Pod 只有一个容器，不需要指定容器：

```
$ kubectl logs mypod
```

而当 Pod 有多个容器的时候，需要指定容器：

```
$ kubectl logs mypod container
```

kubectl logs 默认会打印所有日志，--limit-bytes 参数可以限定日志打印量，而通过--tail 参数可以只打印最新的指定行数的日志，或者通过--since 和--since-time 打印指定时间的日志。另外，通过设置--follow 参数，将实时打印日志流，达到 tail -f 的效果。

kubectl attach

kubectl attach 命令用于连接到 Pod 中启动的容器，类似于 docker attach，如果不指定容器，则选择 Pod 的第一个容器：

```
$ kubectl attach mypod
```

也可以指定容器：

```
$ kubectl attch mypod container
```

kubectl exec

kubectl exec 命令用于在 Pod 的容器中执行命令，类似于 docker exec，如果不指定容器，则选择 Pod 的第一个容器：

```
$ kubectl exec mypod -- date
```

当然可以指定容器执行：

```
$ kubectl exec mypod container -- date
```

提示

kubectl exec 命令需要在 Kubernetes Node 上安装 nsenter。

kubectl port-forward

kubectl port-forward 命令可以为 Pod 设置端口转发，通过在本机监听指定端口，访问这些端口的请求将会被转发到 Pod 的容器中对应的端口上。

执行 kubectl port-forward 命令的时候需要指定 Pod 和端口转发规则，比如 80 端口转发 80 端口，443 端口转发 443 端口：

```
$ kubectl port-forward mypod 80:80 443:443
```

提示

kubectl port-forward 命令需要在 Kubernetes Node 上安装 nsenter 和 socat。

8.3.5 Replication Controller 操作

kubectl run

kubectl run 命令可以用来创建 Replication Controller，创建的时候必须指定容器镜像：

```
$ kubectl run nginx --image nginx
```

创建的 Replication Controller 的 Pod 副本数是 1，可以通过--replicas 参数设置 Pod 的副本数为 2：

```
$ kubectl run nginx --image nginx --replicas 2
```

默认情况下，创建出来的 Replication Controller 会为 Pod 设置一个 Label，Label 的 Key 为 run，Value 为 Replication Controller 的名称。如果 kubectl run 命令中设置了--labels 参数，则会覆盖这个 Label。

kubectl run 命令中只可以设置一个容器，支持容器的属性设置如下所示。

- --command：容器的启动命令。
- --port：容器内部的端口。
- --hostport：容器映射到宿主机的端口。
- --env：容器的环境变量。
- --requests：容器的资源请求规格。
- --limits：容器的资源限制规格。

kubectl scale

kubectl scale 命令可以用来修改 Replication Controller 的 Pod 副本数，即实现 Pod 的弹性伸缩。执行的时候需要指定 Replication Controller 和 Pod 副本数：

```
$ kubectl scale replicationcontroller nginx --replicas=3
```

kubectl scale 命令如果设置了--current-replicas 参数，那么会验证当前 Pod 的副本数是否等于--current-replicas 配置的数目，相等才进行修改操作。

kubectl autoscale

kubectl autoscale 命令可以为 Replication Controller 创建 Horizontal Pod Autoscaler，即实现 Pod 的自动伸缩。执行的时候需要指定 Replication Controller，以及 Pod 的最大和最小副本数：

```
$ kubectl autoscale replicationcontroller nginx --min=1 --max=10
```

8.3.6 Service 操作

kubectl expose

kubectl expose 命令可以用来创建 Service，创建的时候需要指定 Pod、Replication Controller 或者 Service，从中提取 Label 来为新建的 Service 配置 Label Selector：

```
$ kubectl expose pod valid-pod --port=444 --name=frontend
$ kubectl expose replicationcontroller nginx --port=80 --target-port=8000
$ kubectl expose service nginx --port=443 --target-port=8443 --name=nginx-https
```

第 2 部分

Kubernetes 高级篇

第 9 章

Kubernetes 网络

Kubernetes 从 Docker 默认的网络模型中独立出来形成一套自己的网络模型，该网络模型更加适应传统的网络模式，应用能够平滑地从非容器环境迁移到同 Kubernetes 中。本章将对比说明 Kubernetes 和 Docker 的网络模型，然后详细讲解 Kubernetes 网络模型的实现细节。

9.1 Docker 网络模型

Docker 使用 Linux 桥接，在宿主机上虚拟一个 Docker 网桥（docker0），Docker 启动一个容器时会根据 Docker 网桥的网段分配容器的 IP，同时 Docker 网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能通过容器的 IP 直接通信，如图 9-1 所示。

Docker 网桥是宿主机虚拟出来的，并不是真实存在的网络设备，外部网络是无法寻址到的，这也意味着外部网络无法直接访问到容器。如果容器希望能够被外部网络访问到，就需要通过映射容器端口到宿主机（端口映射），即使用 docker run 创建容器时通过–p 或-P 参数来启用，访问容器的时候通过[宿主机 IP]:[容器端口]访问容器。

实际上，端口映射通过在 iptables 的 NAT 表中添加相应的规则，所以我们也将端口映射方式称为 NAT 方式。在早期组建 Docker 机器集群的方案中，往往是选择了 NAT 方式的网络模型。这种网络模型对使用的便利性是有意义的，但并不理想。这个模型需要对各种

端口进行映射，这会限制宿主机的能力，在容器编排上也增加了复杂度。

- 端口是个稀缺资源，这就需要解决端口冲突和动态分配端口问题。这不但使调度复杂化，而且应用程序的配置也将变得复杂，具体表现为端口冲突、重用和耗尽。
- NAT 将地址空间分段的做法引入了额外的复杂度。比如容器中应用所见的 IP 并不是对外暴露的 IP，因为网络隔离，容器中的应用实际上只能检测到容器的 IP，但是需要对外宣称的则是宿主机的 IP，这种信息的不对称将带来诸如破坏自注册机制等问题。

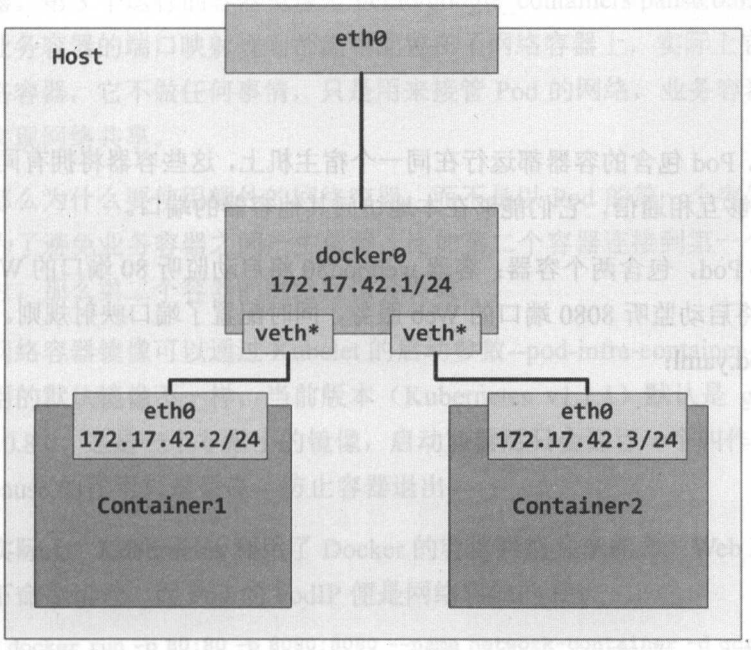


图 9-1 Docker 网络

9.2 Kubernetes 网络模型

Kubernetes 从 Docker 网络模型（NAT 方式的网络模型）中独立出来形成一套新的网络模型。该网络模型的目标是：每一个 Pod 都拥有一个扁平化共享网络命名空间的 IP，称为 PodIP。通过 PodIP，Pod 能够跨网络与其他物理机和 Pod 进行通信。一个 Pod 一个 IP 的（IP-Per-Pod）模型创建了一个干净、反向兼容的模型。在该模型中，从端口分配、网络、

域名解析、服务发现、负载均衡、应用配置和迁移等角度，Pod 都能够被看成虚拟机或物理机，这样应用就能够平滑地从非容器环境（物理机或虚拟机）迁移到同一个 Pod 内的容器环境。

为了实现这个网络模型，在 Kubernetes 中需要解决几个问题：

- 容器间通信（Container to Container）
- Pod 间通信（Pod to Pod）
- Service 到 Pod 的通信（Service to Pod）

9.3 容器间通信

Pod 是容器的集合，Pod 包含的容器都运行在同一个宿主机上，这些容器将拥有同样的网络空间，容器之间能够互相通信，它们能够在本地访问其他容器的端口。

现在创建一个 Web Pod，包含两个容器：容器 webpod80 将启动监听 80 端口的 Web 服务，容器 webpod8080 将启动监听 8080 端口的 Web 服务，同时配置了端口映射规则。Web Pod 的定义文件 web-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: webpod
  labels:
    name: webpod
spec:
  containers:
    - name: webpod80
      image: jonlangemak/docker:web_container_80
      ports:
        - containerPort: 80
          hostPort: 80
    - name: webpod8080
      image: jonlangemak/docker:web_container_8080
      ports:
        - containerPort: 8080
          hostPort: 8080
```

Web Pod 运行成功后，在其所在的 Node 上查询容器：

```
$ docker ps
CONTAINER ID  IMAGE                                  PORTS
63dc7e032ab6  jonlangemak/docker:web_container_8080
4ac1a5156a04  jonlangemak/docker:web_container_80
b77896498f8f  gcr.io/google_containers/pause:0.8.0   0.0.0.0:80->80/tcp,0.0.0.0:8080->8080/tcp
```

可以看到运行了 3 个容器，其中前两个容器是在 Web Pod 定义中指定的，可以称为业务容器。第 3 个运行的容器镜像是 gcr.io/google_containers/pause:0.8.0，Web Pod 定义中设置的业务容器的端口映射规则都集中配置在了网络容器上，实际上它是 Kubernetes 中定义的网络容器，它不做任何事情，只是用来接管 Pod 的网络，业务容器通过加入网络容器的网络实现网络共享。

那么为什么要使用额外的网络容器，而不是以 Pod 的第一个容器作为网络容器呢？主要是为了避免业务容器之间产生依赖，比如第二个容器连接到第一个容器，当第一个容器宕机时，那么第二个容器的网络栈也会失效。

网络容器镜像可以通过 Kubelet 的启动参数--pod-infra-container-image 指定，不同版本下使用的默认镜像不一样，当前版本（Kubernetes v1.1.1）默认是 gcr.io/google_containers/pause:0.8.0，这是一个非常小的镜像，启动容器后只会运行一个叫作 pause 的程序，顾名思义，pause 的作用只是暂停，防止容器退出。

实际上，Kubernetes 利用了 Docker 的容器网络共享能力，Web Pod 中的容器类似于使用以下命令运行，而 Pod 的 PodIP 便是网络容器的 IP：

```
$ docker run -p 80:80 -p 8080:8080 --name network-container -d gcr.io/google_containers/pause:0.8.0
$ docker run --net container:network-container -d jonlangemak/docker:web_container_80
$ docker run --net container:network-container -d jonlangemak/docker:web_container_8080
```

这样一来，Pod 中的所有容器都是互通的，而 Pod 对外可以看成一个完整网络单元，如图 9-2 所示。

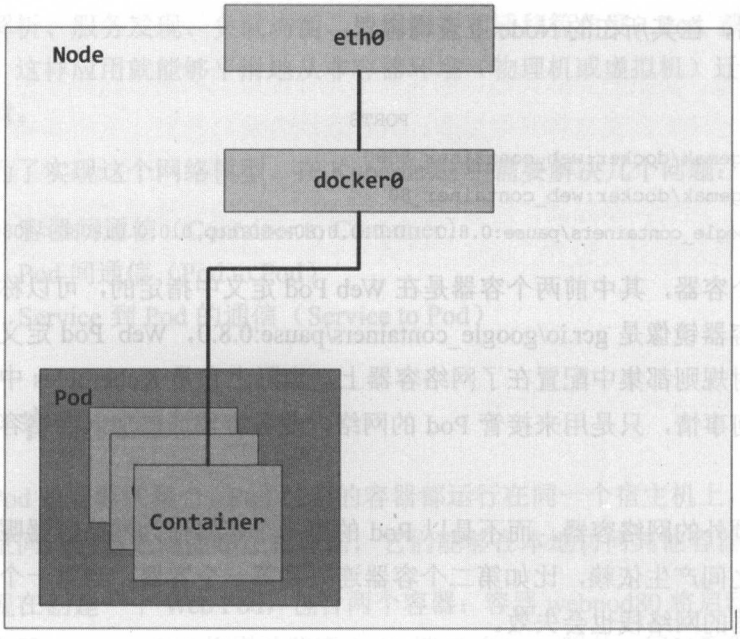


图 9-2　Pod 中的容器网络

9.4　Pod 间通信

Kubernetes 网络模型是一个扁平化的网络平面，在这个网络平面内，Pod 作为一个网络单元同 Kubernetes Node 的网络处于同一层级。

我们考虑一个最小的 Kubernetes 网络拓扑，如图 9-3 所示，在这个网络拓扑中满足以下条件。

- Pod 间通信：Pod1 和 Pod2（同主机），Pod1 和 Pod3（跨主机）能够通信。
- Node 与 Pod 间通信：Node1 和 Pod1/ Pod2（同主机），Pod3（跨主机）能够通信。

那么第一个问题是如何保证 Pod 的 PodIP 是全局唯一的。其实做法也很简单，因为 Pod 的 PodIP 是 Docker 网桥分配的，所以将不同 Kubernetes Node 的 Docker 网桥配置成不同的 IP 网段即可。

另外，同一个 Kubernetes Node 上的 Pod/容器原生能通信，但是 Kubernetes Node 之间的 Pod/容器是如何通信的，这就需要对 Docker 进行增强，在容器集群中创建一个覆盖网络

（Overlay Network），联通各个节点，目前可以通过第三方网络插件来创建覆盖网络，比如 Flannel 和 Open vSwitch 等。

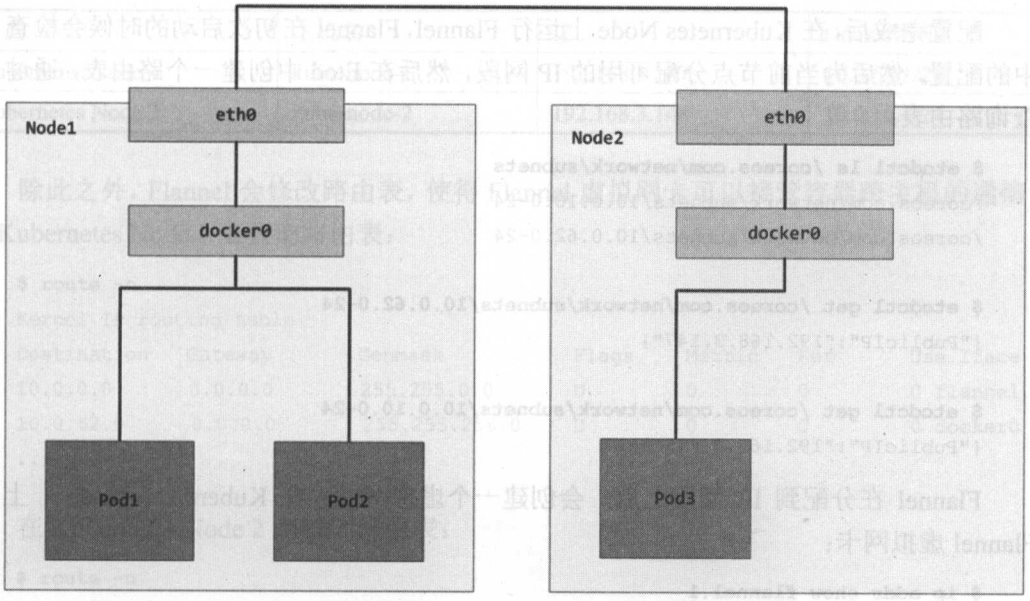


图 9-3 Kubernetes 网络拓扑

9.4.1 Flannel 实现 Kubernetes 覆盖网络

Flannel 是由 CoreOS 团队设计开发的一个覆盖网络工具，它通过在集群中创建一个覆盖网络，为主机设定一个子网，通过隧道协议封装容器之间的通信报文，实现容器的跨主机通信。

现在我们利用 Flannel 联通两个 Kubernetes Node，如表 9-1 所示。

表 9-1 Kubernetes Node 信息

节点	主机名	IP
Kubernetes Node 1	kube-node-1	192.168.3.147
Kubernetes Node 2	kube-node-2	192.168.3.148

Flannel 使用 Etcd 作为配置和协调中心，在运行 Flannel 之前需要在 Etcd 中进行配置。Flannel 的 Etcd 配置目录可以通过启动参数-etcd-prefix 指定，默认是 /coreos.com/network。

首先需要在 Etcd 上配置 Flannel 网络信息：

```
$ etcdctl set /coreos.com/network/config '{ "Network": "10.0.0.0/16" }'
```

配置完成后，在 Kubernetes Node 上运行 Flannel，Flannel 在初次启动的时候会检查 Etcd 中的配置，然后为当前节点分配可用的 IP 网段，然后在 Etcd 中创建一个路由表，通过 Etcd 查询路由表：

```
$ etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/10.0.10.0-24
/coreos.com/network/subnets/10.0.62.0-24

$ etcdctl get /coreos.com/network/subnets/10.0.62.0-24
{"PublicIP":"192.168.3.147"}

$ etcdctl get /coreos.com/network/subnets/10.0.10.0-24
{"PublicIP":"192.168.3.148"}
```

Flannel 在分配到 IP 网段之后，会创建一个虚拟网卡，在 Kubernetes Node 1 上查询 Flannel 虚拟网卡：

```
$ ip addr show flannel.1
5: flannel.1: <BROADCAST, MULTICAST, UP, LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN
    link/ether 62:71:56:28:bd:dd brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.0/16 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::6071:56ff:fe28:bddd/64 scope link
       valid_lft forever preferred_lft forever
```

另外，Flannel 会配置 Docker 网桥（docker0），实际上就是通过修改 Docker 的启动参数--bip 来实现。这样一来，集群中每个节点的 Docker 网桥就分配好了全局唯一的 IP 网段，从而创建出来的容器也将拥有全局唯一的 IP，比如在 Kubernetes Node 1 上查询 Docker 网桥：

```
$ ip addr show docker0
6: docker0: <NO-CARRIER, BROADCAST, MULTICAST, UP> mtu 1450 qdisc noqueue state DOWN
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.1/24 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

由此，Flannel 为两个 Kubernetes Node 划分好了容器网络网段，如表 9-2 所示。

表 9-2 Kubernetes Node 容器网络网段划分（Flannel）

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.0.62.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.0.10.1/24

除此之外，Flannel 会修改路由表，使得 Flannel 虚拟网卡可以接管容器跨主机的通信。在 Kubernetes Node 1 上查询路由表：

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags   Metric   Ref      Use Iface
10.0.0.0        0.0.0.0         255.255.0.0     U       0        0        0 flannel.1
10.0.62.0       0.0.0.0         255.255.255.0   U       0        0        0 docker0
...
```

在 Kubernetes Node 2 上查询路由表：

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags   Metric   Ref      Use Iface
10.0.0.0        0.0.0.0         255.255.0.0     U       0        0        0 flannel.1
10.0.10.0       0.0.0.0         255.255.255.0   U       0        0        0 docker0
...
```

这样一来，当一个节点的容器访问另一个节点的容器时，源节点上的数据会从 docker0 网桥路由到 flannel1.1 网卡，在目的节点会从 flannel 1.1 网卡路由到 docker0 网桥。比如现在有一个数据包要从 IP 为 10.0.62.2 的容器发到 IP 为 10.0.10.2 的容器，根据 Kubernetes Node 1 的路由表，它只与 10.0.0.0/16 这条记录匹配，因此数据包从 docker0 网桥出来以后就被路由到了 flannel1.1 网卡。同理，在 Kubernetes Node 2 上，由于目的地址匹配在 docker0 网桥对于的 10.0.10.0/24 这个记录上，数据包从 flannel1.1 网卡出来以后就被路由到了 docker0 网桥，最后转发给目标容器。

Flannel 虚拟网卡接收到的数据包会被 Flannel 服务进行封装，Flannel 将通过隧道协议封装这些数据包，目前隧道协议已经支持 UDP、VxLAN 等，默认是 UDP。当容器跨主机通信的时候，源主机的 Flannel 服务将接收到的数据包包装在另一种网络包中，然后目的主

机的 Flannel 服务再进行解包。

最终，Flannel 将运行在所有 Kubernetes Node 上，Flannel 重新规划容器集群网络，从而使得集群中所有容器能够获得同属一个内网且不重复的 IP，并让属于不同节点上的容器能够直接通过内网 IP 通信，Flannel 实现的网络结构如图 9-4 所示。

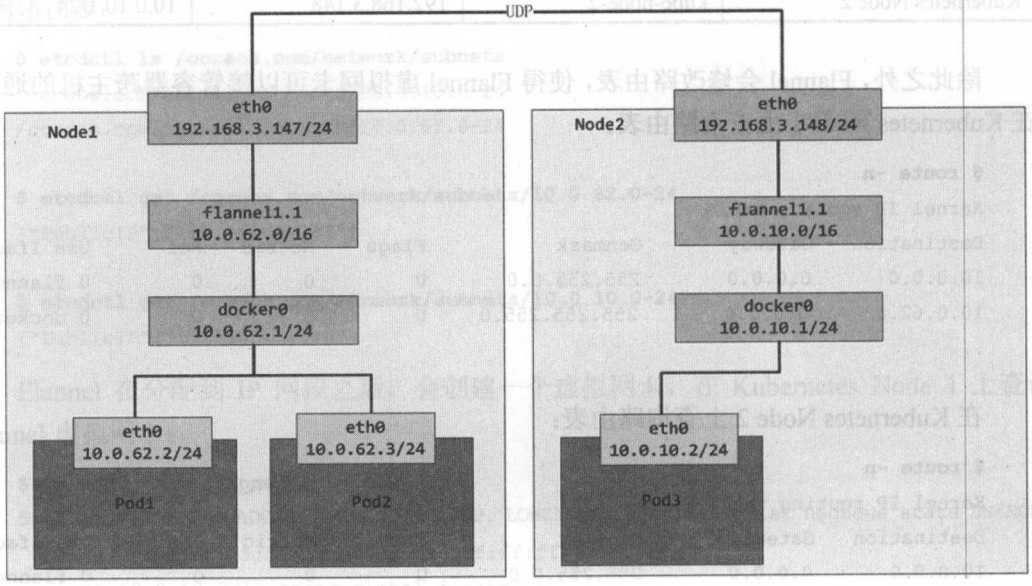


图 9-4　Flannel 实现 Kubernetes 覆盖网络

9.4.2　使用 Open vSwitch 实现 Kubernetes 覆盖网络

Open vSwitch 是一个高质量的、多层虚拟交换机，使用开源 Apache 2.0 许可协议，由 Nicira Networks 开发。它的目的是让大规模网络自动化可以通过编程扩展，同时仍然支持标准的管理接口和协议。

Open vSwitch 也提供了对 OpenFlow 协议的支持，用户可以使用任何支持 OpenFlow 协议的控制器对 Open vSwitch 进行远程管理控制。Open vSwitch 是一项非常重要的 SDN 技术，可以灵活地创建出满足各种需求的虚拟网络，也包括 Kubernetes 中的覆盖网络。

现在我们利用 Open vSwitch 联通两个 Kubernetes Node。为了保证容器 IP 不冲突，所以必须规划好 Kubernetes Node 上 Docker 网桥的网段，如表 9-3 所示。

表 9-3 Kubernetes Node 容器网络网段划分（Open vSwitch）

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.246.0.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.246.1.1/24

Open vSwitch 实现的网络模型如图 9-5 所示。

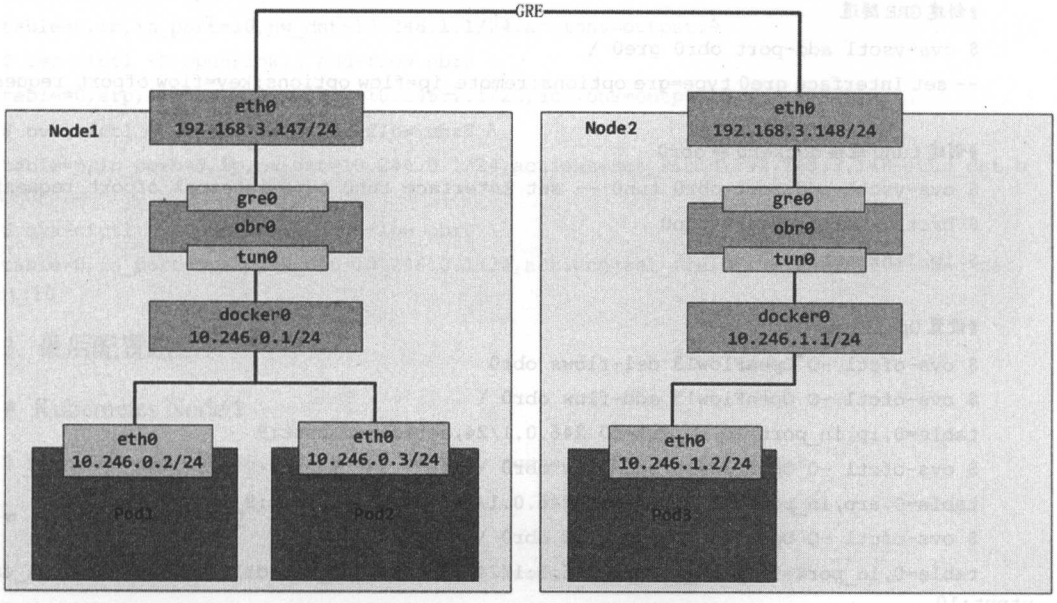


图 9-5 Open vSwitch 实现 Kubernetes 覆盖网络

1. 首先设置 Docker 网桥：

- Kubernetes Node 1

```
$ brctl addbr docker0
$ ip link set dev docker0 up
$ ifconfig docker0 10.246.0.1 netmask 255.255.255.0 up
```

- Kubernetes Node 2

```
$ brctl addbr docker0
$ ip link set dev docker0 up
$ ifconfig docker0 10.246.1.1 netmask 255.255.255.0 up
```

2. 然后创建 Open vSwitch 虚拟网桥 obr0，连接各个网络设备：

- Kubernetes Node 1

```
#创建 obr0
$ ovs-vsctl add-br obr0 -- set Bridge obr0 fail-mode=secure
$ ovs-vsctl set bridge obr0 protocols=OpenFlow13

#创建 GRE 隧道
$ ovs-vsctl add-port obr0 gre0 \
-- set Interface gre0 type=gre options:remote_ip=flow options:key=flow ofport_request=10

#创建 tun0 连接 docker0 和 obr0
$ ovs-vsctl add-port obr0 tun0 -- set Interface tun0 type=internal ofport_request=9
$ brctl addif docker0 tun0
$ ip link set tun0 up

#设置 OpenFlow 规则
$ ovs-ofctl -O OpenFlow13 del-flows obr0
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,ip,in_port=10,nw_dst=10.246.0.1/24,actions=output:9
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,arp,in_port=10,nw_dst=10.246.0.1/24,actions=output:9
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,in_port=9,ip,nw_dst=10.246.1.1/24,actions=set_field:192.168.3.149->tun_dst,o
utput:10
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,in_port=9,arp,nw_dst=10.246.1.1/24,actions=set_field:192.168.3.149->tun_dst,
output:10
```

- Kubernetes Node 2

```
#创建 obr0
$ ovs-vsctl add-br obr0 -- set Bridge obr0 fail-mode=secure
$ ovs-vsctl set bridge obr0 protocols=OpenFlow13

#创建 GRE 隧道
$ ovs-vsctl add-port obr0 gre0 \
-- set Interface gre0 type=gre options:remote_ip=flow options:key=flow ofport_request=10
```

```
#创建 tun0 连接 docker0 和 obr0
$ ovs-vsctl add-port obr0 tun0 -- set Interface tun0 type=internal ofport_request=9
$ brctl addif docker0 tun0
$ ip link set tun0 up

#设置 OpenFlow 规则
$ ovs-ofctl -O OpenFlow13 del-flows obr0
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,ip,in_port=10,nw_dst=10.246.1.1/24,actions=output:9
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,arp,in_port=10,nw_dst=10.246.1.1/24,actions=output:9
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,in_port=9,ip,nw_dst=10.246.0.1/24,actions=set_field:192.168.3.148->tun_dst,o
utput:10
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
table=0,in_port=9,arp,nw_dst=10.246.0.1/24,actions=set_field:192.168.3.148->tun_dst,
output:10
```

3. 最后配置路由：

- Kubernetes Node 1

```
$ ip route add 10.246.0.0/16 dev docker0 scope link src 10.246.0.1
```

- Kubernetes Node 2

```
$ ip route add 10.246.0.0/16 dev docker0 scope link src 10.246.1.1
```

9.5 Service 到 Pod 通信

Service 在 Pod 之间起到服务代理的作用，对外表现为一个单一访问接口，将请求转发给 Pod，Service 的网络转发是 Kubernetes 实现服务编排的关键一环。

现在有一个 Service，查询详情如下：

```
$ kubectl describe service myservice
Name:           myservice
Namespace:      default
Labels:         <none>
Selector:       name=mypod
Type:           ClusterIP
```

```
IP:              10.254.206.148
Port:            http 80/TCP
Endpoints:       10.0.62.87:80,10.0.62.88:80,10.0.62.89:80
Session Affinity: None
No events.
```

可以看到，该 Service 的虚拟 IP 是 10.254.206.148，端口 80/TCP 对应的 Endpoints 包含 3 个后端：10.0.62.87:80、10.0.62.88:80 和 10.0.62.89:80，即当请求 10.254.149.17:80 时，会转发到这些后端之一。

首先虚拟 IP 是由 Kubernetes 创建的，虚拟 IP 的网段是通过 Kubernetes API Server 的启动参数--service-cluster-ip-range=10.254.0.0/16 配置的。

另一个关键组件是 Kubernetes Proxy，Kubernetes Proxy 组件负责实现虚拟 IP 路由和转发，而在容器覆盖网络之上又实现了 Kubernetes 层级的虚拟转发网络。Kubernetes Proxy 需要满足以下功能：

- 转发访问 Service 的虚拟 IP 的请求到 Endpoints。
- 监控 Service 和 Endpoints 的变化，实时刷新转发规则。
- 提供负载均衡能力。

在当前版本（Kubernetes v1.1.1）中，Kubernetes Proxy 有两种实现机制：Userspace 和 Iptables，可以通过 Kubernetes Proxy 的启动参数--proxy-mode 指定。

9.5.1 Userspace 模式

在 Userspace 模式下，Kubernetes Proxy 将会为每一个 Service 在主机上启用随机端口进行监听，并且创建 Iptables 规则重定向访问 Service 虚拟 IP 的请求到这个端口上，而 Kubernetes Proxy 将请求转发到 Endpoints。在此模式下，Kubernetes Proxy 起到反向代理的作用，请求的转发由 Kubernetes Proxy 在用户空间下完成。Kubernetes Proxy 需要监控 Endpoints 的变化，实时刷新转发规则，如图 9-6 所示。

当前 Kubernetes Proxy 只是 3 层（TCP/UDP Over IP）转发，默认的负载均衡策略是轮询方式。

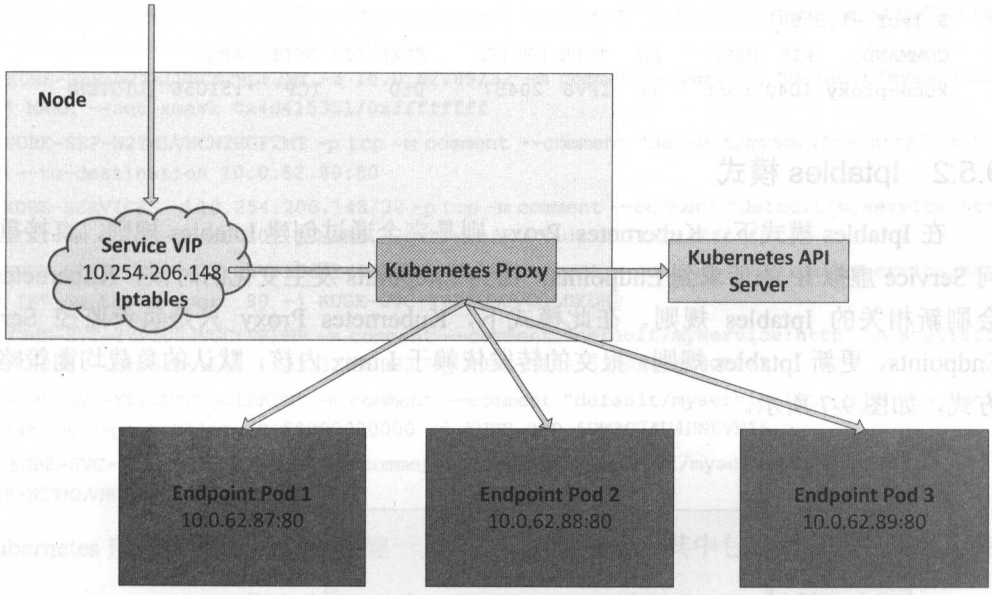


图 9-6 Kubernetes Proxy 的 Userspace 模式

通过 iptables-save 命令可查询关于 Service myservice 的 Iptables 规则：

```
$ iptables-save|grep myservice
    -A KUBE-PORTALS-CONTAINER -d 10.254.206.148/32 -p tcp -m comment --comment
"default/myservice:http" -m tcp --dport 80 -j REDIRECT --to-ports 35841
    -A KUBE-PORTALS-HOST -d 10.254.206.148/32 -p tcp -m comment --comment
"default/myservice:http" -m tcp --dport 80 -j DNAT --to-destination 192.168.3.146:35841
```

Kubernetes Proxy 会为 Service 创建两条 Iptables 规则，其中包含如下两个 Iptables 自定义链。

- KUBE-PORTALS-CONTAINER：用于匹配容器发出的报文，绑定在 NAT 表 PREROUTING 链。
- KUBE-PORTALS-HOST：用于匹配宿主机发出的报文，绑定在 NAT 表 OUTPUT 链。

对于 Service myservice，两条 Iptables 规则的作用都是为了将目的 IP 为 10.254.206.148/32，并且目的端口为 80 的报文重定向到本机的 35841 端口，而 35841 端口就是 Kubernetes Proxy 监听的端口，Kubernetes Proxy 监听在 35841，作为反向代理将请求转发到 Service myservice 的 Endpoints。

```
$ lsof -i:35841
COMMAND    PID USER   FD   TYPE DEVICE    SIZE/OFF NODE NAME
kube-proxy 1040 root   5u   IPv6 20457     0t0      TCP  *:51056 (LISTEN)
```

9.5.2 Iptables 模式

在 Iptables 模式下，Kubernetes Proxy 则是完全通过创建 Iptables 规则，直接重定向访问 Service 虚拟 IP 的请求到 Endpoints。而当 Endpoints 发生变化的时候，Kubernetes Proxy 会刷新相关的 Iptables 规则。在此模式下，Kubernetes Proxy 只是负责监控 Service 和 Endpoints，更新 Iptables 规则，报文的转发依赖于 Linux 内核，默认的负载均衡策略是随机方式，如图 9-7 所示。

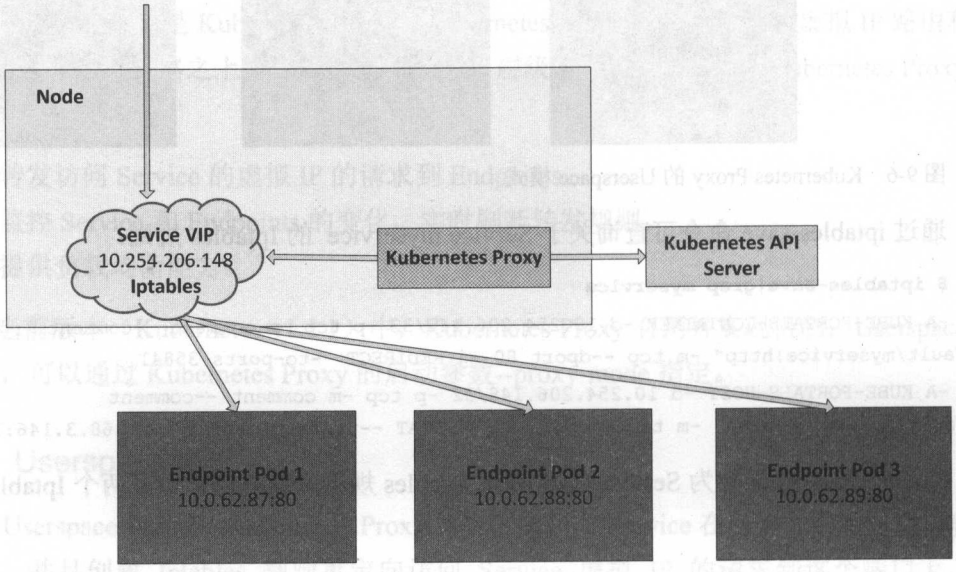


图 9-7 Kubernetes Proxy 的 Iptables 模式

通过 iptables-save 命令可查询到关于 Service myservice 的 Iptables 规则：

```
$ iptables-save|grep myservice
    -A KUBE-SEP-5534AF2Y644GLPZI -s 10.0.62.87/32 -m comment --comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
    -A KUBE-SEP-5534AF2Y644GLPZI -p tcp -m comment --comment "default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.87:80
    -A KUBE-SEP-5UWZC74U4HN6VNI5 -s 10.0.62.88/32 -m comment --comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
```

```
    -A KUBE-SEP-5UWZC74U4HN6VNI5 -p tcp -m comment --comment "default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.88:80
    -A KUBE-SEP-W2TMLVNCN2RGF2MT -s 10.0.62.89/32 -m comment --comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
    -A KUBE-SEP-W2TMLVNCN2RGF2MT -p tcp -m comment --comment "default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.89:80
    -A KUBE-SERVICES -d 10.254.206.148/32 -p tcp -m comment --comment "default/myservice:http
cluster IP" -m tcp --dport 80 -j MARK --set-xmark 0x4d415351/0xffffffff
    -A KUBE-SERVICES -d 10.254.206.148/32 -p tcp -m comment --comment "default/myservice:http
cluster IP" -m tcp --dport 80 -j KUBE-SVC-YZPZUSJVGCLDKDHP
    -A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment "default/myservice:http" -m statistic
--mode random --probability 0.33332999982 -j KUBE-SEP-5534AF2Y644GLPZI
    -A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment "default/myservice:http" -m statistic
--mode random --probability 0.50000000000 -j KUBE-SEP-5UWZC74U4HN6VNI5
    -A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment "default/myservice:http" -j
KUBE-SEP-W2TMLVNCN2RGF2MT
```

Kubernetes Proxy 会为 Service 创建一系列 Iptables 规则，其中包含 Iptables 自定义链。

- KUBE-SERVICES：绑定在 NAT 表 PREROUTING 链和 OUTPUT 链。
- KUBE-SVC-*：代表一个 Service，绑定在 KUBE-SERVICES。
- KUBE-SEP-*：代表 Endpoints 的每一个后端，绑定在 KUBE-SVC-*。

对于 Service myservice，Service 对应的 Iptables 自定义链是 KUBE-SVC-YZPZUSJVGC-LDKDHP，Endpoints 对应的 Iptables 自定义链是 KUBE-SEP-5534AF2Y644GLPZI、KUBE-SEP- 5UWZC74U4HN6VNI5 和 KUBE-SEP-W2TMLVNCN2RGF2MT。

通过 iptables 查询可以更加清晰地看出转发的规则：

```
$ iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
KUBE-SERVICES  all  --  0.0.0.0/0            0.0.0.0/0   /* kubernetes service portals */

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
KUBE-SERVICES  all  --  0.0.0.0/0            0.0.0.0/0  /* kubernetes service portals */
```

```
    Chain POSTROUTING (policy ACCEPT)
    target     prot opt source               destination
    MASQUERADE  all  --  0.0.0.0/0            0.0.0.0/0    /* kubernetes service traffic
requiring SNAT */ mark match 0x4d415351

    Chain KUBE-SERVICES (2 references)
    target     prot opt source               destination
    MARK       tcp  --  0.0.0.0/0            10.254.206.148   /* default/myservice:http
cluster IP */ tcp dpt:80 MARK set 0x4d415351
    KUBE-SVC-YZPZUSJVGCLDKDHP  tcp  --  0.0.0.0/0            10.254.206.148   /* default/
myservice:http cluster IP */ tcp dpt:80

    Chain KUBE-SVC-YZPZUSJVGCLDKDHP (1 references)
    target     prot opt source               destination
    KUBE-SEP-5534AF2Y644GLPZI  all  --  0.0.0.0/0            0.0.0.0/0            /* default/
myservice:http */ statistic mode random probability 0.33332999982
    KUBE-SEP-5UWZC74U4HN6VNI5  all  --  0.0.0.0/0            0.0.0.0/0            /* default/
myservice:http */ statistic mode random probability 0.50000000000
    KUBE-SEP-W2TMLVNCN2RGF2MT  all  --  0.0.0.0/0            0.0.0.0/0            /* default/
myservice:http */

    Chain KUBE-SEP-5534AF2Y644GLPZI (1 references)
    target     prot opt source               destination
    MARK       all  --  10.0.62.87           0.0.0.0/0            /* default/myservice:http */
MARK set 0x4d415351
    DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/myservice:http */
tcp to:10.0.62.87:80

    Chain KUBE-SEP-5UWZC74U4HN6VNI5 (1 references)
    target     prot opt source               destination
    MARK       all  --  10.0.62.88           0.0.0.0/0            /* default/myservice:http */
MARK set 0x4d415351
    DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/myservice:http */
tcp to:10.0.62.88:80

    Chain KUBE-SEP-W2TMLVNCN2RGF2MT (1 references)
    target     prot opt source               destination
    MARK       all  --  10.0.62.89           0.0.0.0/0            /* default/myservice:http */
MARK set 0x4d415351
    DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/myservice:http */
tcp to:10.0.62.89:80
```

第 10 章

Kubernetes 安全

安全永远是一个重大的话题，特别是对于 Kubernetes 这样的云计算平台，更需要设计出一套完善的安全方案，以应对复杂的场景。本章将阐述 Kubernetes 的安全保障机制，主要包括 Kubernetes API 的安全访问、容器安全和多租户。

10.1 Kubernetes 安全原则

Kubernetes 设计出了一套 API 和敏感信息处理方案，也提供了容器安全保障，以下是 Kubernetes 的安全设计原则：

- 保证容器与其运行的宿主机之间有明确的隔离。
- 限制容器对基础设施或者其他容器造成不良影响。
- 最小特权原则——限定每个组件只被赋予执行操作所必需的最小特权，由此确保可能产生的损失最小。
- 普通用户明确区别于系统管理员。
- 能够给普通用户赋予管理权限。
- 应用能够安全地获取敏感信息。

10.2 Kubernetes API 的安全访问

Kubernetes API Server 是 Kubernetes 系统的入口，以 REST API 接口方式提供给外部客户和内部组件调用，对于 Kubernetes API 的访问调用需要进行严格管控，保证系统的安全。

10.2.1 HTTPS

Kubernetes API Server 采用的是 REST 模式的 API 服务，REST 利用传统 Web 特点，提出了一个既适于客户端应用又适于服务端应用的统一架构，极大程度上统一及简化了网站架构设计。

REST 的全称是 Representational State Transfer，表示表述性状态传递，无须 Session，为了安全，每次请求都得带上身份认证信息。REST 是基于 HTTP 的，也是无状态的，HTTP 是明文传输的，所以建议所有的请求都通过 HTTPS 发送，保证对于系统中的重要数据做 SSL 加密传输，如证书、账号密码等。

Kubernetes API Server 支持 HTTP 和 HTTPS，相关启动参数如表 10-1 所示。

表 10-1 Kubernetes API Server 的启动参数

参数	说明
--insecure-bind-address=127.0.0.1	HTTP（不安全）绑定的本机地址，0.0.0.0表示监听本机的所有地址，默认是127.0.0.1
--insecure-port=8080	不安全端口，HTTP（不安全）绑定的本机端口，默认是8080
--bind-address=0.0.0.0	HTTPS（安全）绑定的本机地址，默认是0.0.0.0
--secure-port=6443	安全端口，HTTPS（安全）绑定的本机端口，默认是6443，当为0时，不启动HTTPS
-tls-cert-file=""	HTTPS需要使用的x509证书，和--tls-private-key-file对应。当-tls-cert-file和--tls-private-key-file都为空的时候，会自动生成自签名证书和私钥，生成目录/var/run/kubernetes
--tls-private-key-file=""	HTTPS需要的x509私钥，和--tls-cert-fil对应

Kubernetes API Server开启HTTPS需要准备的HTTPS证书，建议从权威CA机构申请：

```
--tls-cert-file=/path/to/cert
--tls-private-key-file=/path/to/key
--secure-port=6443
--bind-address=0.0.0.0
```

因为 HTTP 是不安全的，所以应该限制 HTTP 的访问，比如设置防火墙规格、防止不信任域的访问，其中最简单的方法是只允许本机访问：

```
--insecure-port=8080
--insecure-bind-address=127.0.0.1
```

10.2.2 认证与授权

Kubernetes API Server 目前支持认证（Authentication）和授权（Authorization）机制进行安全访问控制。认证和授权只作用于 Kubernetes API Server 的安全端口（--secure-port），非安全端口（--insecure-port）不受约束。

10.2.2.1 认证

Basic Authentication

Basic Authentication 是指客户端在使用 HTTP 访问受限资源时，必须使用用户名和密码以获取认证。这是认证中最简单的方法，长期以来，这种认证方法被广泛使用。当通过 HTTP 访问一个使用 Basic Authentication 保护的资源时，服务器通常会在 HTTP 请求的响应中加入一个“401 需要身份验证”的头域，来通知客户提供用户凭证，以使用资源。如果正在使用浏览器访问需要认证的资源，浏览器会弹出一个窗口，让你输入用户名和密码，如果所输入的用户名在资源使用者的验证列表中，并且密码完全正确，此时，用户才可以访问受限的资源。但是这种方式安全性较低，就是简单地将用户名和密码进行 Base64 编码放到头域中。正是因为是 Base64 编码存储，最好使用 HTTPS 进行加密传输。

Kubernetes API Server 开启 Basic Authentication 需要提供一个 CSV 格式的文件用于设置用户列表，每行表示一个用户，共 3 列：密码、用户名和用户 ID，以下是一个示例：

basic_auth.csv

```
admin_passwd,admin,1
test_passwd,test,2
```

然后设置 Kubernetes API Server 的启动参数：

```
--basic-auth-file=/path/to/basic_auth.csv
```

Token Authentication

Token 是一个用户自定义的任意字符串，具有随机性、不可预测性，相比于密码更加安全，一般黑客或软件无法猜测出来。Token Authentication 实际上同 Basic Authentication 类似，使用 Token 替换账号密码，可在一定程度上提高安全性。

Kubernetes API Server 支持 Token Authentication，同样需要提供一个 CSV 格式的文件用于设置用户列表，每行表示一个用户，共 3 列：Token、用户名和用户 ID，以下是一个示例：

token_auth.csv

```
ceGlx8PkwDz0Z5LsOelDShpCD1j67r0a,admin,1
5oEhn1YY6V7J31m3wZX1Tzp8XFXOAHXJ,test,2
```

其中 Token 是任意字符串，可以用以下命令生成：

```
$ echo $(cat /dev/urandom | base64 | tr -d "=+/" | dd bs=32 count=1 2> /dev/null)
```

然后设置 Kubernetes API Server 的启动参数：

```
--token-auth-file =/path/to/token_auth.csv
```

Client Certificate Authentication

Client Certificate 是一种用于证明用户身份的客户端数字证书，如果服务端开启 Client Certificate Authentication，客户端访问的时候就需要提供 Client Certificate。

Kubernetes API Server 支持 Client Certificate Authentication，需要提供信任的客户端 CA 证书，然后配置启动参数：

```
--client-ca-file=/path/to/ca.crt
```

OpenID Authentication

OpenID 是一套以用户为中心的分散式身份认证系统，用户只需注册获取 OpenID 之后，就可以凭借此 OpenID 账号在多个系统之间自由登录使用，而不需要在每一个系统中注册账号，实现用户认证。

Kubernetes API Server 支持 OpenID Authentication，相关启动参数如下：

```
--oidc-issuer-url=<url>
--oidc-client-id=<id_token>
```

Keystone Authentication

Keystone 是 OpenStack 框架中负责管理身份验证、服务规则和服务令牌功能的模块。用户访问资源需要验证用户的身份与权限，服务执行操作也需要进行权限检测，这些都可以通过 Keystone 来处理。

Kubernetes API Server 支持对接 Keystone 来实现认证，相关启动参数如下：

```
--experimental-keystone-url=<AuthURL>
```

10.2.2.2 授权

Kubernetes API Server 在认证之后，通过授权（Authorization）可进一步进行安全访问控制。授权可以通过 Kubernetes API Server 的--authorization-mode 启动参数设置以下几种模式：

- --authorization-mode=AlwaysDeny
- --authorization-mode=AlwaysAllow
- --authorization-mode=ABAC

其中 AlwaysDeny 表示拒绝所有请求，AlwaysAllow 允许所有请求。

ABAC 是一种基于属性的访问控制，根据设置好的访问策略检查请求的属性，不符合访问策略的请求会被拒绝。

API 请求中有 5 个属性可以用作访问控制：

- 用户，请求用于认证的用户，比如使用 Basic Authentication 时，用户需要输入用户名和密码进行认证，那么授权则根据该用户进行判断。
- 用户组，用户所在的用户组，一个用户可以在多个用户组内。
- 请求是否只读，比如 GET 请求都是只读的。
- 请求访问的资源，比如/api/v1/namespaces/default/pods 请求的资源是 pods，一些情况下资源为空，比如/version。
- 资源所在的 Namespace，某些资源不属于任何 Namespace，比如 Node，那么 Namespace 即为空。

当设置为 ABAC 模式时，还需要指定策略文件（--authorization-policy-file），策略文件采用一种 One JSON Object Per Line 的配置格式，即每一行是一个 JSON 格式的策略，用于设置访问控制权限，策略属性如表 10-2 所示。

表 10-2 ABAC 模式的策略属性

属性	类型	说明
user	string	如果指定，需要和请求认证用户匹配
group	string	如果指定，需要和请求认证用户组匹配
readonly	boolean	如果为true，说明只允许GET请求
resource	string	如果指定，需要和请求资源匹配
namespace	string	如果指定，需要和请求资源所在Namespace匹配

策略文件示例：

```
# admin 拥有所有权限
{"user":"admin"}

# scheduler 拥有读 pods 的权限
{"user":"scheduler","readonly": true,"resource": "pods"}

# scheduler 拥有读写 events 的权限
{"user":"scheduler","resource": "events "}

# kubelet 拥有读 pods 的权限
{"user":"kubelet","readonly": true,"resource": "pods"}

# kubelet 拥有读 services 的权限
{"user":"kubelet","readonly": true,"resource": "services"}

# kubelet 拥有读 endpoints 的权限
{"user":"kubelet","readonly": true,"resource": "endpoints"}

# kubelet 拥有读写 events 的权限
{"user":"kubelet","resource": "events"}

# alice 拥有 namespace  projectCaribou 下的所有权限
{"user":"alice","namespace": "projectCaribou"}

# alice 拥有 namespace  projectCaribou 下的读权限
{"user":"bob","readonly": true,"namespace": "projectCaribou"}

# cindy 拥有 namespace  projectCaribou 下的 pods 的读权限
{"user":"cindy","resource": "pods","readonly": true,"namespace": "projectCaribou"}
```

10.2.3 准入控制 Admission Controller

请求在认证和授权之后，Kubernetes 提供了 Admission Controller 进一步对请求进行准入控制。Admission Controller 作为 Kubernetes API Server 的一部分，并以插件的形式存在，不过需要编译进 Kubernetes API Server 可执行程序才能使用。

Kubernetes API Server 将按顺序检查请求，如果其中任何一项没通过，那么就会拒绝请求。

Admission Controller 支持的插件如表 10-3 所示。

表 10-3 Admission Controller 插件

名称	说明
AlwaysAdmit	此插件允许所有的请求
AlwaysDeny	此插件拒绝所有的请求
ServiceAccount	此插件用于支持Service Account
SecurityContextDeny	此插件检查创建Pod的Security Context 是否可用，不可用的话拒绝
ResourceQuota	此插件用于支持Resource Quota，拒绝对于任何超过Resource Quota 的请求
LimitRanger	用于支持Limit Ranger，拒绝不符合LimitRanger约束的请求
NamespaceLifecycle	删除一个Namespace会删除该Namespace下所有的资源（pods、services等）。在Namespace被删除的过程中，此插件将会拒绝任何试图在该Namespace下创建新资源的请求。除此之外，拒绝试图在不存在的Namespace下创建资源的请求

在 Kubernetes API Server 启动的时候，可以配置需要哪些 Admission Controller，以及它们的顺序，建议设置为：

```
--admission_control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccoun
t,ResourceQuota
```

10.3 Service Account

Service Account 概念的引入是基于这样的使用场景：运行在 Pod 里的进程需要调用 Kubernetes API 以及非 Kubernetes API 的其他服务。我们使用 Service Account 来为 Pod 提供认证。

Service Account 和 User Account 可能会带来一定程度上的混淆，它们的区别如下：

- User Account 通常是为人类设计的，而 Service Account 则是为运行在 Pod 中的应用设计的。
- User Account 是全局的，即可以跨 Namespace 使用；而 Service Account 是限定 Namespace 的，即仅在所属的 Namespace 下使用。
- 创建一个新的 User Account 通常需要较高的特权并且需要经过比较复杂的业务逻辑，而 Service Account 则不然。

提示

开启 Service Account 功能，要添加 Service Account Admission Controller，即设置 Kubernetes API Server 的启动参数：

```
--admission_control=...ServiceAccount...
```

Kubernetes API Server 的相关启动参数如表 10-4 所示。

表 10-4 Kubernetes API Server 的启动参数

--service-account-key-file=""	配置一个包含PEM-encoded x509 RSA的私钥或者公钥，用于验证Service Account Token

Kubernetes Controller Manager 的相关启动参数如表 10-5 所示。

表 10-5 Kubernetes Controller Manager 的启动参数

--service-account-private-key-file=""	配置一个包含PEM-encoded x509 RSA的私钥，用于签发Service Account Token
--root-ca-file=""	配置访问Kubernetes API Server的CA 证书，将赋值给Service Account Secret

10.3.1 使用默认 Service Account

Kubernetes Controller Manager 会为每个 Namespace 默认创建一个 Service Account，我们称为默认 Service Account：

```
$ kubectl get serviceaccount
NAME      SECRETS   AGE
default   1         1d
```

默认 Service Account 包含一个 Secret：

```
$ kubectl describe serviceaccount default
Name:                default
Namespace:           default
Labels:              <none>

Mountable secrets:   default-token-7t9ja

Tokens:              default-token-7t9ja
```

```
Image pull secrets:    <none>
```

查询该 Secret：

```
$ kubectl describe secret default-token-7t9ja
Name:         default-token-7t9ja
Namespace:    default
Labels:       <none>
Annotations: kubernetes.io/service-account.name=default,kubernetes.io/service-
account.uid=...

Type:    kubernetes.io/service-account-token

Data
====
token:     ...
ca.crt:  1838 bytes
```

查询显示这是一个 kubernetes.io/service-account-token 类型的 Secret，称为 Service Account Secret，包含两个数据 token 和 ca.crt，这也是由 Kubernetes Controller Manager 生成的。其中 token 是由--service-account-private-key-file 指定的密钥签发生成的，而 ca.crt 是由 --root-ca-file 指定的 CA 证书。

这样一来，新建的 Pod 如果没有指定 Service Account，就会使用默认 Service Account，挂载 Service Account Secret 到容器目录中（/var/run/secrets/kubernetes.io/serviceaccount），Pod 中的应用通过 token 和 ca.crt 访问 Kubernetes API Server。同时 Kubernetes 提供非常方便的方式让 Pod 获取 Kubernetes API Server 的地址。

Kubernetes 在初始化的时候会创建一个 Kubernetes Service，叫作 kubernetes，它代表的就是 Kubernetes API Server：

```
$ kubectl describe service kubernetes
Name:              kubernetes
Namespace:         default
Labels:            component=apiserver, provider=kubernetes
Selector:          <none>
Type:              ClusterIP
IP:                10.254.0.1
Port:     https    443/TCP
```

```
Endpoints:         192.168.3.146:6443
Session Affinity:  None
No events.
```

Kubernetes Service 的虚拟 IP 是 10.254.0.1，这是 Kubernetes API Server 配置的 Service 网段的第一个 IP（--service-cluster-ip-range=10.254.0.0/16），然后将会转发 HTTPS 的 443 端口到 192.168.3.146:6443，即 Kubernetes API Server 的地址和 HTTPS 安全端口。并且 Kubernetes Service 是最早创建的，新建的 Pod 中可以通过环境变量获取来访问 Kubernetes API Server（也可以通过 DNS 方式）：

```
KUBERNETES_SERVICE_HOST=10.254.0.1
KUBERNETES_PORT_443_TCP=tcp://10.254.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.254.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.254.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
```

Pod 中的进程就可以访问 Kubernetes API Server，我们现在简单实现一个脚本 curl-k8s-api.sh 来调用 Kubernetes API：

```
#!/bin/sh

# curl-k8s-api.sh

Endpoint=$1
ServiceAccountToken=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
ServiceAccountRootCA=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
KubernetesServerURL="https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT"

curl  -XGET -H "Authorization: Bearer $ServiceAccountToken" \
--cacert $ServiceAccountRootCA \
${KubernetesServerURL}${Endpoint}
```

脚本 curl-k8s-api.sh 读取 Service Account 的 token 和 ca.crt，通过环境变量生成 URL，然后通过 curl 命令调用 Kubernetes API Server。我们需要将脚本 curl-k8s-api.sh 放入 Pod 的容器中，然后访问/api 获取版本信息：

```
$ kubectl exec pod -- curl-k8s-api.sh /api
{
```

```
  "versions": [
    "v1"
  ]
}
```

另外，Service Account 会自动创建一个认证用户，用户的命名格式是：

```
system:serviceaccount:[namespace]:[serviceaccountname]
```

比如对于在 Namespace my-ns 中，默认 Service Account 对应的用户名是：

```
system:serviceaccount:my-ns:default
```

如果 Kubernetes API Server 设置了 ABAC 的授权模式，需要为 Service Account 设置授权策略，否则会导致 Pod 无法通过 Service Account 访问 Kubernetes API Server，比如设置为只读权限：

```
{"user":"system:serviceaccount: my-ns:default", "readonly": true}
```

10.3.2 创建自定义 Service Account

用户可以创建自定义的 Service Account，Service Account 的定义文件 serviceaccount.yaml：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-serviceaccount
```

通过定义文件创建 Service Account：

```
$ kubectl create -f serviceaccount.yaml
serviceaccount "my-serviceaccount" created
```

查询 Service Account 的详细信息：

```
$ kubectl describe serviceaccount my-serviceaccount
Name:          my-serviceaccount
Namespace:     default
Labels:        <none>

Image pull secrets:    <none>

Mountable secrets:     my-serviceaccount-token-ivsml

Tokens:                my-serviceaccount-token-ivsml
```

可以看到，该 Service Account 包含一个 Service Account Secret，Kubernetes Controller Manager 会保证每个 Service Account 都至少包含一个 Service Account Secret。如果不希望使用自动创建的 Service Account Secret，可以手工创建，Service Account Secret 的定义文件 serviceaccountSecret.yaml：

```
apiVersion: v1
kind: Secret
metadata:
  name: my-serviceaccount-secret
  annotations:
    kubernetes.io/service-account.name: my-serviceaccount
type: kubernetes.io/service-account-token
```

通过定义文件创建 Service Account Secret：

```
$ kubectl create -f serviceaccountSecret.yaml
secret "my-serviceaccount-secret" created
```

添加创建好的 Service Account Secret 到 Service Account：

```
$ kubectl patch serviceaccount my-serviceaccount \
-p '{"secrets": [{"name": "my-serviceaccount-secret"}]}'
"my-serviceaccount" patched
```

然后删除自动创建的 Service Account Secret：

```
$ kubectl delete secret my-serviceaccount-token-ivsm1
secret "my-serviceaccount-token-ivsm1" deleted
```

这样就成功替换了 Service Account Secret：

```
$ kubectl describe serviceaccount my-serviceaccount
Name:        my-serviceaccount
Namespace:           default
Labels:      <none>

Image pull secrets:    <none>

Mountable secrets:     my-serviceaccount-secret

Tokens:                my-serviceaccount-secret
```

在 Pod 的定义中可以通过.spec.serviceAccountName 指定 Service Account，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: busybox
    command:
      - sleep
      - "3600"
  restartPolicy: Always
  serviceAccountName: my-serviceaccount
```

10.3.3 Service Account 添加 Image Pull Secret

在 Service Account 中添加 Image Pull Secret 时，当 Pod 使用该 Service Account 的时候就自动关联上其中的 Image Pull Secret；而当为默认 Service Account 添加 Image Pull Secret 时，那么 Pod 会自动关联上 Image Pull Secret，不必显示配置。

首先创建一个 Image Pull Secret，可参考 4.3.1 节：

```
$ kubectl get secret myregistrykey
NAME            TYPE                             DATA      AGE
myregistrykey   kubernetes.io/dockercfg   1         1h
```

添加 Image Pull Secret 到默认 Service Account：

```
$ kubectl patch serviceaccounts default \
-p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
"default" patched
```

修改默认 Service Account 成功后，可以查询到 Image Pull Secret 添加成功：

```
$ kubectl describe serviceaccounts default
Name:        default
Namespace:   default
Labels:      <none>
```

```
Image pull secrets:    myregistrykey

Mountable secrets:     default-token-mymp9

Tokens:                default-token-mymp9
```

这样一来，新创建的 Pod 都会自动关联上 Image Pull Secret：

```
spec:
  imagePullSecrets:
  - name: myregistrykey
```

10.4 容器安全

10.4.1 Linux Capability

Docker 对于容器中的 root 用户是进行限制的，这是基于 Linux 内核的 Capability 机制实现的。Linux Capability 将内核系统资源访问划分为许多的权限属性，从而可以对权限进行精细化管理。

对于 Docker 容器中的 root 用户，很多系统相关的操作权限都是被剥夺的，只具备超级用户的一些基本权限。如果需要授予 Docker 容器足够的权限，可以使用特权模式，即设置 docker run 的参数--privileged=ture。

在 Pod 的定义中可以通过.spec.containers[].securityContext.privileged=true 设置容器的特权模式：

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-server
  labels:
    role: nfs-server
spec:
  containers:
    - name: nfs-server
      image: jsafrane/nfs-data
```

```
    ports:
      - name: nfs
        containerPort: 2049
    securityContext:
      privileged: true
```

注意

Pod 中的容器要设置特权模式，需要设置 Kubernetes API Server 和 Kubelet 的启动参数--allow-privileged=ture 来允许开启容器的特权模式。

当 Docker 容器设置了特权模式之后，那么 Docker 容器的 root 权限将得到大幅度提升。由于 Docker 容器与宿主机处于共享同一个内核操作系统的状态，因此 Docker 容器将完全拥有内核的管理权限，这就存在安全隐患。

而且对应用程序来说，往往只需要特定的权限，比如一个程序需要使用 ping 命令，这是一个 SUID 命令，会以 root 权限运行，而实际上这个程序只是需要 RAW 套接字建立必要 ICMP 数据包，除此之外的其他 root 权限对这个程序都是没有必要的。

Docker 支持添加和删除容器的 Linux Capability，即 docker run 命令的--cap-add 和--cap-drop 参数。在 Pod 的定义中可以通过.spec.containers[].securityContext.capabilities.add 和.spec.containers[].securityContext.capabilities.drop 添加和删除容器的 Linux Capability：

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-server2
  labels:
    role: nfs-server
spec:
  containers:
    - name: nfs-server
      image: nginx
      ports:
        - name: nfs
          containerPort: 2049
      securityContext:
        capabilities:
          add:
```

```
    - IPC_LOCK
    - NET_ADMIN
  drop:
    - SETGID
```

10.4.2 SELinux

SELinux（Security Enhanced Linux）是 Linux 内核的安全模块，提供了一种灵活的强制访问控制系统。SELinux 拥有一个灵活而强制性的访问控制结构，旨在提高 Linux 系统的安全性，提供强健的安全保证。

Docker 支持使用 SELinux 进行安全加固，通过 SELinux 可以根据类别和 MCS/MLS 等级来区分进程，保护宿主机不受容器影响。

docker run 中通过--security-opt 可以修改容器的 MCS/MLS 标签：

```
--security-opt="label:user:USER"
--security-opt="label:role:ROLE"
--security-opt="label:type:TYPE"
--security-opt="label:level:LEVEL"
```

相应的，在 Pod 的定义中可按如下方式修改容器的 MCS/MLS 标签:

```
securityContext:
  seLinuxOptions:
    user: USER
    role: ROLE
    type: TYPE
    level: LEVEL
```

10.5 多租户

多租户是云计算中的一个重要能力，是云计算集中式的数据中心，以服务的形式提供给用户。多租户是共享和隔离互相作用下的产物，用户需要处在不同的租户下，同租户内部是共享的，但是不同租户之间是隔离的。

Kubernetes 中的 Namespace 就是租户的概念，Kubernetes 整个平台的内容通过 Namespace 划分为多个逻辑平面，不同个人或者团队在不同 Namespace 中共享 Kubernetes，Namespace 之间互相不感知。

Kubernetes 以 Namespace 作为管理单位，可以控制安全访问策略，设置资源配额等，为此需要使用 Kubernetes 规划好 Namespace，比如可以根据功能、区域、部门或者业务等。

我们可以像其他 API 对象一样创建 Namespace，Namespace 定义文件 development-namespace.yaml：

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

通过定义文件创建 Namespace：

```
$ kubectl create -f development-namespace.yaml
namespace "development" created
```

创建成功后可以查询 Namespace：

```
$ kubectl describe namespace development
Name:    development
Labels:  name=development
Status:  Active

No resource quota.

No resource limits.
```

第 11 章

Kubernetes 资源管理

资源管理是 Kubernetes 的一个关键能力，Kubernetes 需要为应用分配足够的资源，又要防止应用无限制使用资源，随着应用规模数量级的增加，这些问题就显得至关重要。本章将阐述 Kubernetes 资源管理的模型和具体实现方法，包括资源分配策略和配额限制等。

11.1 Kubernetes 资源模型

虚拟化技术是云计算平台的基础，其目标是对计算资源进行整合或划分，这是云计算管理平台中的关键技术。虚拟化技术为云计算管理平台的资源管理提供了资源调配上的灵活性，从而使得云计算管理平台可以通过虚拟化层整合或划分计算资源。

相比于虚拟机，新出现的容器技术使用了一系列的系统级别的机制，诸如利用 Linux Namespace 进行空间隔离，通过文件系统的挂载点决定容器可以访问哪些文件，通过 Cgroup 确定每个容器可以利用多少资源。此外，容器之间共享同一个系统内核，这样当同一个内核被多个容器使用时，内存的使用效率会得到提升。

容器和虚拟机两大虚拟化技术，虽然实现方式完全不同，但是它们的资源需求和模型其实是类似的。容器像虚拟机一样需要内存、CPU、硬盘空间和网络带宽，宿主机系统可以将虚拟机和容器都视作一个整体，为这个整体分配其所需的资源，并进行管理。当然，虚拟机提供了专用操作系统的安全性和更牢固的逻辑边界，而容器在资源边界上比较松散，这带来了灵活性以及不确定性。

Kubernetes 是一个容器集群管理平台，Kubernetes 需要统计整体平台的资源使用情况，合理地将资源分配给容器使用，并且要保证容器生命周期内有足够的资源来保证其运行。更进一步，如果资源发放是独占的，即资源已发放给了一个容器，同样的资源不会发放给另外一个容器，对于空闲的容器来说占用着没有使用的资源（比如 CPU）是非常浪费的，Kubernetes 需要考虑如何在优先度和公平性的前提下提高资源的利用率。

11.2 资源请求和限制

计算资源是 Pod 或者容器运行所需的，包括：

- CPU

单位是核（core）。

```
cpu: 1          #1 核
cpu: 0.25       #0.25 核
cpu: 250m       #0.25 核
```

- 内存（Memory）

单位是字节（byte）。

```
memory: 1024  #1024 Byte
memory: 512Ki #512 KByte
memory: 256Mi #256 MByte
memory: 1.5Gi #1.5 GByte
```

创建 Pod 的时候，可以指定每个容器的资源请求（Request）和资源限制（Limit），资源请求是容器所需的最小资源需求，资源限制则是容器不能超过的资源上限，它们的大小关系必须是：

```
0 <= request <= limit <= Infinity
```

在容器的定义中，资源请求通过 resources.requests 设置，资源限制通过 resources.limits 设置，目前可以指定的资源类型只有 CPU 和内存。资源请求和限制是可选配置，默认值根据是否设置 Limit Range 而定。如果资源请求没有指定也没有默认值，那么资源请求就等于资源限制。

以下定义的 Pod 包含两个容器：第一个容器的资源请求是 0.5 核 CPU 和 256 MByte 内

存，资源限制是 1 核 CPU 和 512 MByte 内存；第二个容器的资源请求是 0.25 核 CPU 和 128MByte 内存，资源限制是 1 核 CPU 和 512 MByte 内存：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "256Mi"
        cpu: "500m"
      limits:
        memory: "512Mi"
        cpu: "1000m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "512Mi"
        cpu: "1000m"
```

Pod 的资源请求/限制是 Pod 中所有容器资源请求/限制的和，比如该 Pod 的资源请求是 0.75 核 CPU 和 384MByte 内存，资源限制是 2 核 CPU 和 1024MByte 内存。

Kubernetes 在调度 Pod 的时候，Pod 的资源请求是调度的一个关键指标。Kubernetes 会获取 Kubernetes Node 的最大资源容量(通过 cAdvisor 接口)，并计算出已使用的资源情况，比如 Node 能够容纳 2 核 CPI 和 2GByte 内存，Node 上已经运作了 4 个 Pod，共请求 1.5 核 CPU 和 1GByte 内存，剩余 0.5 核 CPU 和 1GByte 内存。Kubernetes Scheduler 调度 Pod 的时候会检查 Node 上是否还有足够资源来满足 Pod 的资源请求，不满足则将该 Node 排除。

资源请求能够保证 Pod 有足够的资源来运行，而资源限制则是防止某个 Pod 无限制地使用资源，导致其他 Pod 崩溃。特别是在公有云场景，往往会有恶意软件通过抢占内存来

攻击平台。

Docker 容器是使用 Linux Cgroup 来实现资源限制的，docker run 就提供了参数对 CPU 和内存进行限制。

- --memory
  docker run 通过--memory 给一个容器可用的内存配额，Cgroup 会限制容器的内存使用，一旦超过配额，容器将会被终止。
  Kubernetes 中 Docker 容器的--memory 值就是 resources.limits.memory 的值，比如 resources.limits.memory=512Mi，那么--memory 的值就是 512*1024*1024*1024。
- --cpu-shares
  docker run 通过--cpu-shares 设置一个容器的可用的 CPU 配额。需要特别注意的是，这是一个相对权重，与实际的处理速度无关。每个新的容器默认将有 1024 CPU 配额，当我们单独讲它的时候，这个值并不意味着什么。但是如果启动两个容器并且两个都将使用 100%的 CPU，CPU 时间将在这两个容器之间平均分割，因为它们两个都有同样的 CPU 配额。如果我们设置容器的 CPU 配额是 512，相对于另外一个 1024 CPU 配额容器，它将使用 1/3 的 CPU 时间。但这不意味着它仅仅能使用 1/3 的 CPU 时间。如果另外一个容器（1024 CPU 配额的）是空闲的，其他容器将被允许使用 100% CPU。对于 CPU 来说，难以清楚地说明多少 CPU 被分配给了哪个容器，这取决于实际的运行情况。

Kubernetes 中 Docker 容器的--cpu-shares 值通过 resources.requests.cpu 或者 resources.requests.cpu 乘以 1024 而得。如果指定 resources.requests.cpu，那么--cpu-shares 就等于 resources.requests.cpu 乘以 1024，如果没有指定 resources.requests.cpu，但是指定了 resources.limits.cpu，那么--cpu-shares 就等于 resources.limits.cpu 乘以 1024，如果 resources.limits.cpu 和 resources.limits.cpu 都没有指定，那么 resources.resources.cpu 就取最小值（当前版本最小值为 2），具体如下：

```
if resources.requests.cpu is definded
  cpuShares=resources.requests.cpu * 1024
else if resources.requests.cpu is undefined
  if resources.limits.cpu is defined
    cpuShares=resources.limits.cpu * 1024
else if resources.limits.cpu is defined
    cpuShares = minShares
```

比如 resources.requests.cpu=250m，那么--cpu-share 的值就是 250m*1024=256。

11.3 Limit Range

Limit Range 设计的初衷是为了满足以下场景：

- 能够约束租户的资源需求。
- 能够约束容器的资源请求范围。
- 能够约束 Pod 的资源请求范围。
- 能够指定容器的默认资源限制。
- 能够指定 Pod 的默认资源限制。
- 能够约束资源请求和限制之间的比例。

提示

Kubernetes 要开启 Limit Range 功能，首先要添加 Limit Range Admission Controller，即设置 Kubernetes API Server 的启动参数：

```
--admission_control=...LimitRange...
```

Limit Range 包含两种类型的设置：Container 和 Pod，包含约束和默认值的配置，如表 11-1 和表 11-2 所示。

表 11-1 Limit Range Container 配置

类型	Container
资源类型	memory cpu
约束	**min：min <= Request (required) <= Limit (optional)** **max：Limit (required) <= max** **maxLimitRequestRatio：maxLimitRequestRatio <= (Limit (required,non-zero) / Request (required,non-zero)**
默认值	default：Limit的默认值 defaultRequest：Request的默认值

表 11-2 Limit Range Pod 配置

类型	Pod
资源类型	memory cpu
约束	**min：Min <= Request (required) <= Limit (optional)** **Max：Limit (required) <= Max** **maxLimitRequestRatio：(Limit (required,non-zero) / Request (required,non-zero) maxLimitRequestRatio**
默认值	Pod的默认值无须直接配置，根据容器的默认值而得

创建 Limit Range 的时候需要注意以下几点。

- **配置数值的时候需要满足以下条件**

```
Min (if specified) <= DefaultRequest (if specified) <= Default (if specified) <= Max (if specified)
```

- **默认值行为**

当 Default 未设置：

```
if LimitRangeItem.Default[resourceName] is undefined
  if LimitRangeItem.Max[resourceName] is defined
    LimitRangeItem.Default[resourceName] = LimitRangeItem.Max[resourceName]
```

当 DefaultRequest 未设置：

```
if LimitRangeItem.DefaultRequest[resourceName] is undefined
  if LimitRangeItem.Default[resourceName] is defined
    LimitRangeItem.DefaultRequest[resourceName] = LimitRangeItem.Default[resourceName]
  else if LimitRangeItem.Min[resourceName] is defined
    LimitRangeItem.DefaultRequest[resourceName] = LimitRangeItem.Min[resourceName]
```

当在 Namespace 下创建 Limit Range 后，就可以设置 Pod 或者容器的资源请求和限制默认值，更重要的功能是对 Pod 和容器的资源规格配置进行约束。现在我们在 Namespace development 下创建一个 Limit Range，Limit Range 的定义文件 limits.yaml：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: development
```

```
spec:
  limits:
  - type: Pod
    max:
      cpu: 4
      memory: 2Gi
    min:
      cpu: 250m
      memory: 8Mi
    maxLimitRequestRatio:
      cpu: 4
      memory: 4
  - type: Container
    default:
      cpu: 500m
      memory: 512Mi
    defaultRequest:
      cpu: 250m
      memory: 256Mi
    max:
      cpu: 2
      memory: 1Gi
    min:
      cpu: 250m
      memory: 8Mi
    maxLimitRequestRatio:
      cpu: 2
      memory: 2
```

通过定义文件创建 Limit Range：

```
$ kubectl create -f limits.yaml
limitrange "limits" created
```

创建成功，查询 Limit Range：

```
$ kubectl describe limitranges limits --namespace=development
Name:         limits
Namespace:    development
Type          Resource Min      Max      Request       Limit     Limit/Request
----          -------- ---      ---      -------       -----     -------------
```

```
Pod          cpu      250m     4        -          -          4
Pod          memory   8Mi      2Gi      -          -          4
Container    cpu      250m     2        250m       500m       2
Container    memory   8Mi      1Gi      256Mi      512Mi      2
```

对于以上内容有以下几点说明。

- 默认值

1. 一个容器的默认资源请求是 256Mbyte 内存和 250m 核 CPU。

2. 一个容器的默认资源限制是 512Mbyte 内存和 500m 核 CPU。

- 约束

1. 一个 Pod 的资源请求和限制必须满足：

```
250m <= requests.cpu <= limits.cpu <= 4
8Mi <= requests.memory <= limits.memory <= 2Gi
limits.memory/requests.memory <= 4
limits.cpu/requests.cpu <= 4
```

2. 一个容器的资源请求和限制必须满足：

```
250m <= requests.cpu <= limits.cpu <= 2
8Mi <= requests.memory <= limits.memory <= 1Gi
limits.memory/requests.memory <= 2
limits.cpu/requests.cpu <= 2
```

至此，在 Namespace development 中创建的 Pod 都需要满足以上约束，比如创建一个 Pod，Pod 的定义文件 test-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  namespace: development
spec:
  containers:
  - name: container1
    image: nginx
    resources:
      requests:
```

```
        memory: 512Mi
        cpu: "999m"
      limits:
        memory: "512Mi"
        cpu: "2"
  - name: container2
    image: nginx
    resources:
      requests:
        memory: 512Mi
        cpu: "250m"
      limits:
        memory: "1Gi"
        cpu: "500m"
  - name: container3
    image: nginx
    resources:
      requests:
        memory: 8ki
        cpu: "250m"
      limits:
        memory: "1Gi"
        cpu: "500m"
```

创建 Pod 会失败：

```
$ kubectl create -f test-pod.yaml
Error from server: error when creating "test-pod.yaml": Pod "test" is forbidden: [
Maximum memory usage per Pod is 2Gi, but limit is 2684354560.,
cpu max limit to request ratio per Container is 2, but provided ratio is 2.002002.,
Minimum memory usage per Container is 8Mi, but request is 8Ki.,
memory max limit to request ratio per Container is 2, but provided ratio is 131072.000000.]
```

其中有 4 个错误：

1. Pod 的最大内存为 2Gi，但是创建的 Pod 资源限制是 1Gi+1Gi+512Mi=2684354560。
2. 容器的 CPU 限制和请求比例不能超过 2，容器 container1 的 CPU 限制和请求比例为 2/999m=2.002002。
3. 容器的最小内存是 8mi，容器 container3 的 CPU 请求是 8Ki。

4. 容器的内存限制和请求比例不能超过 2，容器 container3 的内存限制和请求比例为 500m /8ki =131072.000000。

11.4 Resource Quota

Kubernetes 是一个多租户架构，当多用户或者团队共享一个 Kubernetes 系统的时候，系统管理员需要防止租户的资源强占，定义好资源分配策略。比如 Kubernetes 系统共有 20 核 CPU 和 32GByte 内存，分配给 A 租户 5 核 CPU 和 16 GByte，分配给 B 租户 5 核 CPU 和 8GByte，预留 10 核 CPU 和 8GByte 内存。这样，租户中所使用的 CPU 和内存的总和不能超过指定的资源配额，促使其更合理地使用资源。

Kubernetes 中提供 API 对象 Resource Quota（资源配额）来实现资源配额，Resource Quota 不仅可以作用于 CPU 和内存，另外还可以限制比如创建 Pod 的数目。Resource Quota 支持的类型如表 11-3 和表 11-4 所示。

- 计算资源配额

表 11-3 计算资源配额

资源名称	说明
cpu	CPU配额
memory	内存配额

限制计算资源的使用，Namespace 中所有 Pod 的资源请求总和不能超过配额，比如 Namespace A 的内存配额为 1GByte，那么 Namespace A 中所有 Pod 的 Memory 请求总和，即 resources.requests.memory 的总和不能超过 1GByte。

- Kubernetes API 对象资源配额

表 11-4 Kubernetes API 对象资源配额

资源名称	说明
pods	Pod的总数目
services	Service的总数目
replicationcontrollers	Replication Controller的总数目
resourcequotas	Resource Quota的总数目
secrets	Secret的总数目
persistentvolumeclaims	Persistent Volume Claim的总数目

限制 Kubernetes API 对象的创建数目，Namespace 中 API 对象的创建数目不能超过配额，比如限制 Namespace A 的 Pod 和 Service 的创建数目。

> **提示**
> Kubernetes 要开启 Resource Quota 支持，首先要添加 Resource Quota Admission Controller，即设置 Kubernetes API Server 的启动参数：
>
> ```
> --admission_control=...ResourceQuota...
> ```

默认情况下，Namespace 是没有 Resource Quota 的，需要另外创建 Resource Quota。一般情况下，一个 Namespace 下配置一个 Resource Quota 即可，但是可以创建多个 Resource Quota，将按多个 Resource Quota 最小值作为配额值。比如一个 Resource Quota 设置 Pod 的数目配额为 10，另一个 Resource Quota 设置 Pod 的数目配额为 5，那么 Pod 的数目不能超过 5。

现在我们在 Namespace development 下创建一个 Resource Quota，Resource Quota 的定义文件 quota.yaml：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
  namespace: development
spec:
  hard:
    cpu: "20"
    memory: 50Gi
    persistentvolumeclaims: "10"
    pods: "10"
    replicationcontrollers: "20"
    resourcequotas: "1"
    secrets: "10"
    services: "5"
```

通过定义文件创建 Resource Quota：

```
$ kubectl create -f quota.yaml
resourcequota "quota" created
```

创建成功后可以查询 Resource Quota，其中包括配额值和使用值：

```
$ kubectl describe resourcequota quota --namespace=development
Name:                   quota
Namespace:              development
Resource                Used    Hard
--------                ----    ----
cpu                     0       20
memory                  0       1Gi
persistentvolumeclaims  0       10
pods                    0       10
replicationcontrollers  0       20
resourcequotas          1       1
secrets                 1       10
services                0       5
```

一旦 Namespace 有 Resource Quota，创建 Pod 的时候就必须指定资源请求，否则 Pod 会创建失败（Kubernetes 返回 403 FORBIDDEN）。同样的，Pod 请求的资源超过了资源配额也会创建失败（Kubernetes 将返回 403 FORBIDDEN）。

现在在 Namespace development 下创建一个 Pod，请求 0.25 核 CPU 和 128MByte 的内存：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: development
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "512Mi"
        cpu: "500m"
```

在 Pod 创建成功后，可以查询到相应的资源使用情况，共消耗 1 个 Pod，以及 0.25 核 CPU 和 128MByte 的内存：

```
$ kubectl describe resourcequota quota --namespace=development
Name:                   quota
Namespace:              development
Resource                Used        Hard
--------                ----        ----
cpu                     250m        20
memory                  134217728   1Gi
persistentvolumeclaims  0           10
pods                    1           10
replicationcontrollers  0           20
resourcequotas          1           1
secrets                 1           10
services                0           5
```

第 12 章

管理和运维 Kubernetes

Kubernetes 向下接管底层资源，向上托管业务应用，要管理和运维这样一套系统可以说是一个巨大的挑战。幸运的是，Kubernetes 已经提供了一些很好的支持来帮助管理 Kubernetes。本章将介绍 Kubernetes 的高可靠性方案、平台监控、平台日志等。

12.1 Daemon Pod

从系统管理的需求来说，我们常常需要在 Kubernetes 的所有节点上运行一些守护进程（Daemon），比如日志收集、监控等。传统的做法是使用一些 Init 工具（比如 init、upstartd，或者 systemd）来进行管理，而 Kubernetes 中支持以 Pod 的形式运行这些守护进程，我们称为 Daemon Pod，实现的方式包括 Static Pod 和 Daemon Set。

12.1.1 Static Pod

Static Pod 是直接由 Kubelet 组件创建、运行在 Node 上的 Pod，Kubelet 组件负责 Static Pod 的持续运行，而无须 Replication Controller 进行关联管理。这样一来，Static Pod 就同 Kubelet 进行绑定，从而运行在 Node 上作为 Daemon Pod。

Static Pod 的创建是通过在 Kubelet 指定的 Manifest 目录中放入 Pod 的定义文件（JSON 或者 YAML 格式）进行的，Manifest 目录是通过 Kubelet 的启动参数--config 配置的。本书 Kubernetes 运行环境指定 Kubelet 的 Manifest 目录为/etc/kubernetes/manifests。

现在通过 Static Pod 在 Kubernetes Node 上运行 Prometheus，一个开源的服务监控系统，可以实现对 Docker 容器进行监控。这需要在所有 Node 上的 Manifest 目录中创建 Static Pod 的定义文件 prometheus-node-exporter.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: prometheus-node-exporter
  labels:
    daemon: prom-node-exp
spec:
  containers:
  - name: c
    image: prom/prometheus
    ports:
    - name: serverport
      containerPort: 9090
      hostPort: 9090
```

Kubelet 将在 Node 上运行 Static Pod，其中 Static Pod 的名称是 Pod 名称拼接上 Node 的名称：[pod_name]-[node_name]：

```
$ kubectl get pod --selector daemon=prom-node-exp --output wide
NAME                                    READY     STATUS    RESTARTS   AGE       NODE
prometheus-node-exporter-kube-node-1    1/1       Running   0          15s       kube-node-1
prometheus-node-exporter-kube-node-2    1/1       Running   0          18s       kube-node-2
prometheus-node-exporter-kube-node-3    1/1       Running   0          25s       kube-node-3
```

如果删除该 Static Pod，Kubelet 会重新创建 Static Pod，保证其持续运行，而只有在 Manifest 目录中删除该 Static Pod 的定义文件，Static Pod 才会被删除。

除了直接在 Manifest 目录中放入 Static Pod 定义文件，还可以通过 Kubelet 的启动参数 --manifest-url=<URL>指定远程 URL，Kubelet 将定期下载 Static Pod 的定义文件进行创建或者更新。

使用 Static Pod 能够运行 Daemon Pod，但是 Static Pod 的管理是比较低效的。比如发生变更，就可能需要修改每个 Node 上的 Kubelet 配置。为此，Kubernetes 提供了一个更加强大的机制来管理运行 Daemon Pod。

12.1.2 Daemon Set

Kubernetes 提供了 Daemon Set，用来在 Kubernetes Node 上创建运行 Daemon Pod。Daemon Set 的作用同 Replication Controller 类似，它们都是管理控制 Pod 的，只不过 Daemon Set 是保证所有（或者部分）Node 上都能够运行 Daemon Pod。

> **提示**
>
> 在当前版本（Kubernetes v1.1.1）中，Daemon Set 是 Beta 测试阶段，需要设置 Kubernetes API Server 启动参数--runtime-config=extensions/v1beta1/daemonsets=true 来开启 Daemon Set 支持。

现在通过 Daemon Set 在 Kubernetes Node 上运行 Prometheus，Daemon Set 的定义文件 prometheus-node-exporter.yaml：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: prometheus-node-exporter
spec:
  template:
    metadata:
      labels:
        daemon: prom-node-exp
    spec:
      containers:
      - name: c
        image: prom/prometheus
        ports:
        - name: serverport
          containerPort: 9090
          hostPort: 9090
```

可以看出，Daemon Set 的定义和 Replication Controller 类似，通过.spec.template 设置 Pod 的模板。因为 Daemon Pod 需要持续运行，所以 Pod 的模板中的重启策略只能是 Always。

在默认情况下，Daemon Set 会在所有 Node 上运行 Daemon Pod，通过.spec.template.spec.nodeSelector 设置 Node Selector，可以只在匹配的 Node 上运行 Daemon Pod。

通过定义文件创建 Daemon Set：

```
$ kubectl create -f prometheus-node-exporter.yaml --validate=false
daemonset "prometheus-node-exporter" created

$ kubectl get daemonset prometheus-node-exporter
NAME                    CONTAINER(S)  IMAGE(S)          SELECTOR              NODE-SELECTOR
prometheus-node-exporter  c           prom/prometheus  daemon=prom-node-exp  <none>
```

Daemon Set 创建成功后，将在所有 Node 上运行 Pod：

```
$ kubectl get pods --selector daemon=prom-node-exp --output wide
NAME                              READY   STATUS    RESTARTS   AGE     NODE
prometheus-node-exporter-qpta1    1/1     Running   0          50s     kube-node-1
prometheus-node-exporter-j04ks    1/1     Running   0          50s     kube-node-2
prometheus-node-exporter-ss23x    1/1     Running   0          50s     kube-node-3
```

当删除 Daemon Set 后，相关联的 Pod 也会被删除：

```
$ kubectl delete daemonset prometheus-node-exporter
daemonset "prometheus-node-exporter" deleted

$ kubectl get pods --selector daemon=prom-node-exp -o wide
NAME     READY     STATUS    RESTARTS   AGE       NODE
```

如果希望删除 Daemon Set 而保留关联的 Pod，运行 kubectl delete 时加上参数 --cascade=false 即可。

12.2 Kubernetes 的高可用性

高可用性是一个老生常谈的话题，当然是因为这是人们非常关心的特性，它决定了一个系统的最终价值。在生产环境中，任何系统都需要持续可靠地运行，保持其服务的高度可用性，这是一个最基本的要求。特别是对于 Kubernetes 这样的云平台，一个将要承载着大量应用的系统，它的任何故障都可能大面积地影响业务，其重要性自然不言而喻。

Kubernetes 属于典型的主从分布式架构，Kubernetes 的重要数据集中存储在 Etcd，数据层的可靠性至关重要，对 Etcd 进行集群化是必不可少的（可参考 14.3.2 节）。

Kubernetes Node 作为 Pod 的运行机，原生支持集群化扩展来提供容灾容错能力。Kubernetes Node 运行后将会注册到 Kubernetes Master，并定时上报心跳信息以说明其可用。

Kubernetes Master 调度 Pod 到可用的 Kubernetes Node 部署运行，如果有 Kubernetes Node 发生宕机，Kubernetes Master 会将该 Kubernetes Node 设置为不可用状态。然后 Replication Controller 会重新创建 Pod，从而调度到新的 Kubernetes Node 上。当然，Kubernetes Node 的数目至少要大于 2 才可以保障 Pod 的高可用性。

在 Kubernetes Master 的高可用性方案中需要特别注意，Kubernetes API Server 可以多实例运行，而 Kubernetes Scheduler 和 Kubernetes Controller Manager 不能有多个实例同时运行。Kubernetes 官方提供了一种方案，如图 12-1 所示。

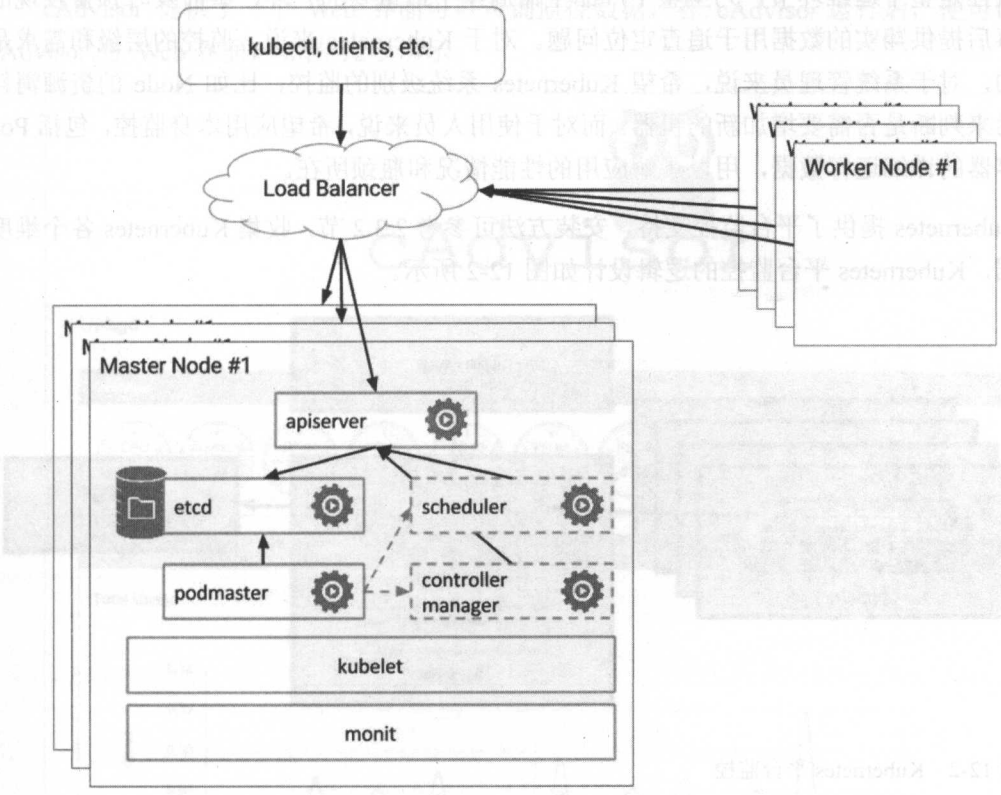


图 12-1 Kubernetes 的高可用性

在这个方案中，Kubernetes Master 多节点部署，其中 Kubernetes API Server、Kubernetes Scheduler 和 Kubernetes Controller Manager 3 个组件分别作为 Static Pod 由 Kubelet 来控制管理。Kubernetes API Server 可以在每个 Kubernetes Master 节点上运行，前端通过负载均衡器作为访问入口。

另外，每个 Kubernetes Master 节点上需要运行一个小程序 Podmaster，Podmaster 通过 Etcd 进行选举，从多个 Kubernetes Master 中选择主节点，只有主节点上会运行 Kubernetes Scheduler 和 Kubernetes Controller Manager。当主节点发生宕机时，选择新的主节点运行 Kubernetes Scheduler 和 Kubernetes Controller Manager。

12.3 平台监控

监控是整个运维环节，乃至整个产品生命周期中最重要的一环，事前及时预警发现故障，事后提供翔实的数据用于追查定位问题。对于 Kubernetes 来说，监控的层级和需求是多样的，对于系统管理员来说，希望 Kubernetes 系统级别的监控，比如 Node 的资源消耗数据用来判断是否需要增加新的机器。而对于使用人员来说，希望应用本身监控，包括 Pod 或者容器的详细运行数据，用以了解应用的性能情况和瓶颈所在。

Kubernetes 提供了平台监控支持，安装方法可参考 2.3.2 节，收集 Kubernetes 各个维度的数据，Kubernetes 平台监控的逻辑设计如图 12-2 所示。

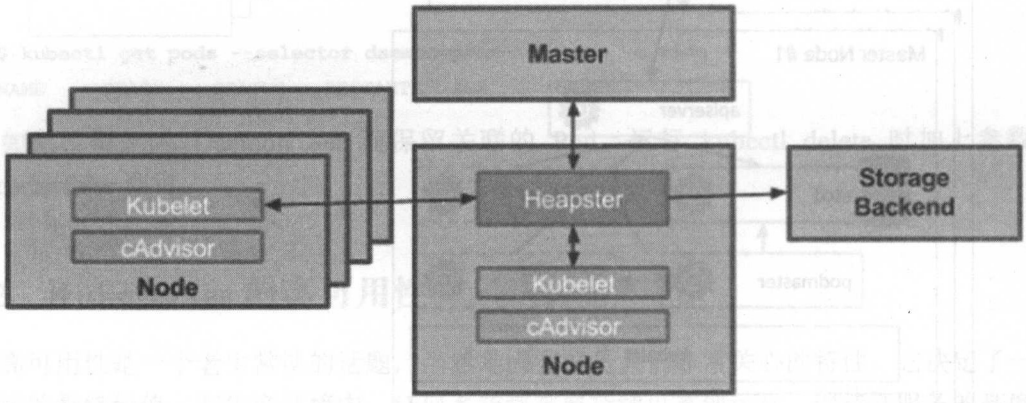


图 12-2 Kubernetes 平台监控

Kubernetes 平台监控的两个关键组件是 cAdvisor 和 Heapster。

12.3.1 cAdvisor

cAdvisor（Container Advisor）是谷歌开源的一个容器监控工具。它是一个守护进程，收集、统计和处理宿主机和容器的数据，包括实时和历史的 CPU、内存、磁盘、网络使用

情况。同时，cAdvisor 提供良好的 Web 界面和 Rest API 来展示数据，帮助系统管理者清楚了解容器的资源使用情况和运行性能数据。

提示

在 cAdvisor 的认识中，容器的概念是广义的，不仅仅包括 Docker 容器，也包括其他容器实现，另外宿主机本身也是一种容器，称为根容器，它是一种原生容器。

Web UI

cAdvisor 提供了一个 Web 界面可以查询监控数据，在 cAdvisor 运行后，便可以访问 cAdvisor 的 Web 界面，如图 12-3 所示。

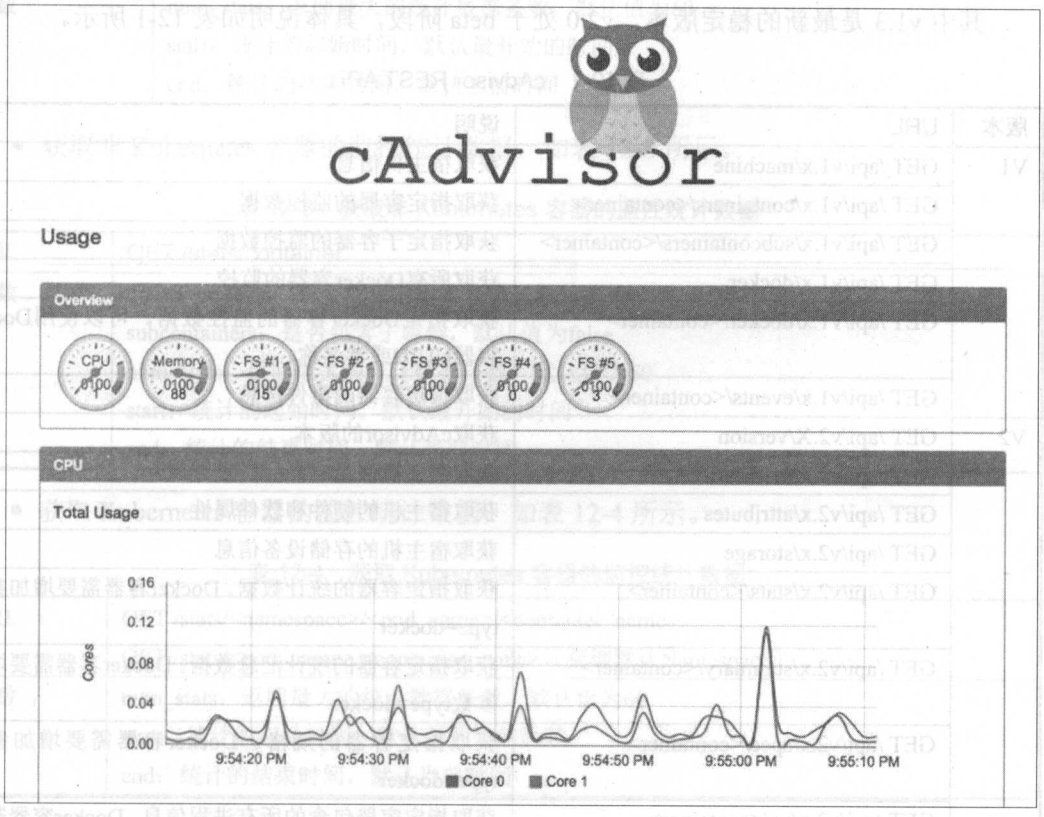


图 12-3 cAdvisor Web 界面

REST API

cAdvisor 提供 REST API 用于获取监控数据，cAdvisor 的 REST API 访问方式为 http://<hostname>:<port>/api/<version>/<request>。

cAdvisor REST API 支持的版本和请求类型如下所示。

- v1.0：containers,machine
- v1.1：containers,machine,subcontainers
- v1.2：containers,docker,machine,subcontainers
- v1.3：containers,docker,events,machine,subcontainers
- v2.0：appmetrics,attributes,events,machine,ps,spec,stats,storage,summary,version

其中 v1.3 是最新的稳定版本，v2.0 处于 beta 阶段，具体说明如表 12-1 所示。

表 12-1　cAdvisor REST API

版本	URL	说明
V1	GET /api/v1.x/machine	获取宿主机信息
	GET /api/v1.x/containers/<container>	获取指定容器的监控数据
	GET /api/v1.x/subcontainers/<container>	获取指定子容器的监控数据
	GET /api/v1.x/docker	获取所有Docker容器的监控
	GET /api/v1.x/docker/<container>	获取指定Docker容器的监控数据。可以使用Docker容器UUID或者名称
	GET /api/v1.x/events/<container>	获取指定容器的事件信息
V2	GET /api/v2.X/version	获取cAdvisor的版本
	GET /api/v2.x/machine	获取宿主机信息
	GET /api/v2.x/attributes	获取宿主机的硬件和软件属性
	GET /api/v2.x/storage	获取宿主机的存储设备信息
	GET /api/v2.x/stats/<container>	获取指定容器的统计数据。Docker容器需要增加参数type=docker
	GET /api/v2.x/summary/<container>	获取指定容器的统计汇总数据。Docker容器需要增加参数type=docker
	GET /api/v2.x/spec/<container>	获取指定容器的规格。Docker容器需要增加参数type=docker
	GET /api/v2.x/ps/<container>	获取指定容器包含的所有进程信息。Docker容器需要增加参数type=docker
	GET /api/v2.x/appmetrics/<container>	支持自定义应用的监控维度，通过API获取监控数据

Kubelet 集成

Kubelet 组件已经集成了 cAdvisor，在 Kubernetes Node 上可以直接访问 cAdvisor，cAdvisor 运行端口可以通过 Kubelet 的启动参数--cadvisor-port 设置，默认是 4194。

Kubelet 基于 cAdvisor 提供了 REST API（默认端口 10255）来获取 Node 和 Pod/容器的监控数据。

- 获取 Node 的监控统计数据，如表 12-2 所示。

表 12-2 获取 Node 的监控统计数据

URL	GET /stats/
参数	num_stats：返回最大的统计数据条数，默认值为60 start：统计的起始时间，默认最开始的时间 end：统计的结束时间，默认当前时间

- 获取非 Kubernetes 容器的监控统计数据，如表 12-3 所示。

表 12-3 获取非 Kubernetes 容器的监控统计数据

URL	GET /stats/container
参数	containerName：需要统计的容器名称，默认值为“/” subcontainers：是否包含子容器，默认值为false num_stats：返回最大的统计数据条数，默认值为60 start：统计的起始时间，默认最开始的时间 end：统计的结束时间，默认当前时间

- 获取 Kubernetes 容器的监控统计数据，如表 12-4 所示。

表 12-4 获取 Kubernetes 容器的监控统计数据

URL	GET /stats/<namespace>/<pod_name>/<container_name> GET /stats/<pod_name>/<container_name>（查询默认Namespace）
参数	num_stats：返回最大的统计数据条数，默认值为60 start：统计的起始时间，默认最开始的时间 end：统计的结束时间，默认当前时间

Kubelet 统计 API 返回的监控统计数据的格式如下所示。

- name：容器名称。

- aliases：容器别名。
- namespace：命名空间，由 cAdvisor 定义。
- subcontainers：包含的子容器列表。
- spec：容器规格，包含如下信息。
  - creation_time：创建时间。
  - labels：包含的标签。
  - has_cpu：是否有 CPU。
  - cpu：CPU 信息。
  - has_memory：是否有内存。
  - memory：内存信息。
  - has_network：是否有网络。
  - has_filesystem：是否有文件系统。
  - has_diskio：是否有磁盘 IO。
  - image：容器镜像。
- stats：容器监控统计数据，每条数据包含如下信息。
  - timestamp：时间戳。
  - cpu：CPU 统计数据。
  - diskio：磁盘 IO 统计数据。
  - memory：内存统计数据。
  - network：网络统计数据。
  - filesystem：文件系统统计数据。
- task_stats：任务统计数据。

12.3.2 Heapster

Heapster 是谷歌开源的容器集群的监控收集工具，它可以集成 Kubernetes 进行监控数据的收集汇总，提供 REST API 来获取 Kubernetes 各个维度的监控数据。另外，Heapster 支持对接第三方系统，将监控数据导入到第三方系统进行进一步处理。

REST API

Heapster 提供 REST API，如表 12-5 所示。

表 12-5 Heapster REST API

类型	URL	说明
metric	GET /api/v1/metric-export	获取最新的Metric数据
	GET /api/v1/metric-export-schema	获取Metric数据的规格
Sink	POST /api/v1/sinks	配置当前的Sink
	GET /api/v1/sinks	获取当前的Sink

集成 Kubernetes

Heapster 在与 Kubernetes 集成的时候，Heapster 调用 Kubernetes API 获取所有 Node 列表，然后调用 Kubelet 的 API 收集汇总监控数据。同时，Heapster 根据收集到的数据建立一个 Kubernetes 监控模型（Metric Model），包括 Kubernetes 以下层级。

- Cluster：整个 Kubernetes 运行环境的监控模型。
- Node：各个 Node 的监控数据模型，包括机器本身的监控和 Node 上运行的 Pod 的监控。
- Namespace：各个 Namespace 监控模型，相当于 Namespace 下所有 Pod 的监控。
- Pod：各个 Pod 的监控模型。
- Container：各个 Container 监控模型。

Kubernetes 监控模型提供 API 来获取各个层级的监控数据，如表 12-6 所示。

表 12-6 Kubernetes 监控模型

层级	URL
Cluster	/api/v1/model/
	/api/v1/model/metrics/
	/api/v1/model/metrics/<metric-name>
	/api/v1/model/stats/
Node	/api/v1/model/nodes/
	/api/v1/model/nodes/<node-name>/
	/api/v1/model/nodes/<node-name>/pods/
	/api/v1/model/nodes/<node-name>/metrics/
	/api/v1/model/nodes/<node-name>/metrics/<metric-name>
	/api/v1/model/nodes/<node-name>/stats/

续表

层级	URL
Namespace	/api/v1/model/namespaces/
	/api/v1/model/namespaces/<namespace-name>/
	/api/v1/model/namespaces/<namespace-name>/metrics/
	/api/v1/model/namespaces/<namespace-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/stats/
Pod	/api/v1/model/namespaces/<namespace-name>/pods/
	/api/v1/model/namespaces/<namespace-name>/pods/{pod-name}/
	/api/v1/model/namespaces/<namespace-name>/pods/{pod-name}/metrics/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/stats/
Container	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/
	/api/v1/model/namespaces/{namespace-name}/pods/<pod-name>/containers/<container-name>/metrics/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/stats/
	/api/v1/model/nodes/<node-name>/freecontainers/
	/api/v1/model/nodes/<node-name>/freecontainers/<container-name>/
	/api/v1/model/nodes/<node-name>/freecontainers/<container-name>/metrics/
	/api/v1/model/nodes/{node-name}/freecontainers/{container-name}/metrics/{metric-name}
	/api/v1/model/nodes/{node-name}/freecontainers/{container-name}/stats/

12.4 平台日志

Kubernetes 提供了平台日志支持，安装方法可参考 2.3.3 节。平台日志基于 Fluentd+Elasticsearch+Kibana，其中 Fluentd 作为 Logging Agent 收集日志汇总到 Elasticsearch。

Fluentd 是一个开源的日志收集系统，支持 150 多个插件，能够将日志收集到 MongoDB、

Redis、Amazon S3 和 Elasticsearch 等；Fluentd 能够以 JSON 格式处理日志，具备每天收集 5000+台服务器上 5TB 的日志数据，每秒处理 50000 条消息的性能。

Fluentd 运行在 Kubernetes 的所有节点上，收集 Kubernetes 组件的日志，Fluentd 的配置如下所示：

```
<source>
  type tail
  format none
  path /var/log/docker.log
  pos_file /var/log/es-docker.log.pos
  tag docker
</source>

<source>
  type tail
  format none
  path /var/log/etcd.log
  pos_file /var/log/es-etcd.log.pos
  tag etcd
</source>

<source>
  type tail
  format none
  path /var/log/kubelet.log
  pos_file /var/log/es-kubelet.log.pos
  tag kubelet
</source>

<source>
  type tail
  format none
  path /var/log/kube-apiserver.log
  pos_file /var/log/es-kube-apiserver.log.pos
  tag kube-apiserver
</source>

<source>
```

```
    type tail
    format none
    path /var/log/kube-controller-manager.log
    pos_file /var/log/es-kube-controller-manager.log.pos
    tag kube-controller-manager
  </source>

  <source>
    type tail
    format none
    path /var/log/kube-scheduler.log
    pos_file /var/log/es-kube-scheduler.log.pos
    tag kube-scheduler
  </source>
```

Fluentd 配置中的 Source 用于指定日志输入资源，其中字段含义如下所示。

- type：指定日志的输入方式，其中 tail 方式是不停地从源文件中获取新的日志。
- format：指定日志的输出格式。
- path：指定日志文件的位置。
- tag：指定日志 tag，用来对不同的日志进行分类。

可以看出，Fluentd 将监控每个组件的日志文件，然后设置上 tag，最后输出。另外，Fluentd 同时会监控容器的日志文件：

```
  <source>
    type tail
    path /var/log/containers/*.log
    pos_file /var/log/es-containers.log.pos
    time_format %Y-%m-%dT%H:%M:%S
    tag kubernetes.*
    format json
    read_from_head true
  </source>
```

Fluentd 会监控/var/log/containers 目录下的所有日志文件，然后以 JSON 格式输出，其中设置的 tag 是 kubernetes.*，在 tail 方式下会被设置为 kubernetes.path.to.file。

实际上，/var/log/containers 目录是由 Kubelet 组件创建的，Kubelet 在其中建立 Pod 的容器日志文件，这些是容器中应用打印到标准输出（Stdout）的日志，即通过 kubectl logs

或者 docker logs 查询到的日志。

比如我们运行的 Hello World Pod，其信息查询如下：

```
$ kubectl describe pod hello-world
Name:          hello-world
Namespace:     default
Image(s):      ubuntu:14.04
Node:          kube-node-2/192.168.3.148
Start Time:    Fri, 04 Dec 2015 19:28:21 +0800
Labels:        <none>
Status:        Succeeded
Reason:
Message:
IP:
Replication Controllers:<none>
Containers:
 hello:
   Container ID:
docker://8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a
   Image:          ubuntu:14.04
   Image ID:
docker://8251da35e7a79dca688682f6da6148a06d358c6f094020844468a782842c2172
   State:          Terminated
     Reason:       Error
     Exit Code:    0
     Started:      Fri, 04 Dec 2015 19:28:28 +0800
     Finished:     Fri, 04 Dec 2015 19:28:29 +0800
   Ready:          False
   Restart Count: 0
   Environment Variables:
......
```

Hello World Pod 包含的一个容器 hello 将输出 Hello World，这条输出日志实际上是由 Docker 进行存储，并通过 Kubernetes 获取的，而 Fluentd 将会进行日志收集。

```
$ kubectl logs hello-world hello
Hello World
```

Docker 容器的日志都会由 Docker 进行存储，而 Kubelet 会在/var/log/containers 下生成一个文件软连接 Docker 存储的容器日志文件，文件格式是[pod_name]_[namespace]_

[container_name]-[container_id].log：

```
$ cat hello-world_default_hello-8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465-c3c9e97a09b8a.log
{"log":"Hello World\n","stream":"stdout","time":"2015-12-04T11:28:29.016916036Z"}
```

那么 Fluentd 将读取这个日志文件，对应的 tag 是：

```
kubernetes.var.log.containers.hello-world_default_hello-8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a.log
```

Fluentd 最终将日志导入到 Elasticsearch，通过检索 Elasticsearch 可以查询到该日志：

```
{
"_index": "logstash-2015.12.04",
"_type": "fluentd",
"_id": "AVFs0tILXx_4Q0Dgz43x",
"_score": 6.7633038,
"_source": {
   "log": "Hello World\n",
   "stream": "stdout",
   "docker": {
      "container_id": "8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a"
   },
   "kubernetes": {
      "namespace": "default",
      "pod_name": "hello-world",
      "container_name": "hello"
   },
   "tag":
"kubernetes.var.log.containers.hello-world_default_hello-8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a.log",
   "@timestamp": "2015-12-04T11:28:29+00:00"
}
}
```

12.5 垃圾清理

Kubernetes 系统在长时间运行后，Kubernetes Node 会下载非常多的镜像，其中可能会

存在很多过期的镜像。同时因为运行大量的容器，容器退出后就变成死亡容器，将数据残留在宿主机上，这样一来，过期镜像和死亡容器都会占用大量的硬盘空间。如果硬盘空间被用光，可能会发生非常糟糕的情况，甚至会导致硬盘的损坏。为此，Kubelet 会进行垃圾清理工作，即定期清理过期镜像和死亡容器。

12.5.1 镜像清理

镜像清理的策略是当硬盘空间使用率超过阈值的时候开始执行，Kubelet 执行清理的时候优先清理最久没有被使用的镜像。

硬盘空间使用率的阈值通过 Kubelet 的启动参数--image-gc-high-threshold 和--image-gc-low-threshold 指定。

12.5.2 容器清理

Kubelet 容器清理的相关参数如表 12-7 所示。

表 12-7 Kubelet 容器清理参数

参数	Kubelet启动参数	说明
MinAge	--minimum-container-ttl-duration	死亡容器能够被删除的最小TTL，默认是1分钟
MaxPerPodContainer	--maximum-dead-containers-per-container	每个Pod允许存在的最大死亡容器数目，默认是2
MaxContainers	--maximum-dead-containers	允许存在的最大死亡容器数目，默认是100

Kubelet 定时执行容器清理，每次根据以上 3 个参数选择死亡容器删除，通常情况下优先删除创建时间最久的死亡容器。Kubelet 不会删除非 Kubelet 管理的容器。

12.6 Kubernetes 的 Web 界面

Kubernetes 提供了一个 Web 界面 Kube UI（https://github.com/kubernetes/kube-ui），用来图形化展示运行环境信息。安装方法可参考 2.3.4 节。安装成功后可以通过 Kubernetes API Server 的接口 http://<kubernetes-master>/ui 进行访问。

Kube UI 首页图形化展示了所有 Kubernetes Node 的资源使用情况，包括 CPU、Memory 和 Filesystem 的使用情况，如图 12-4 所示。

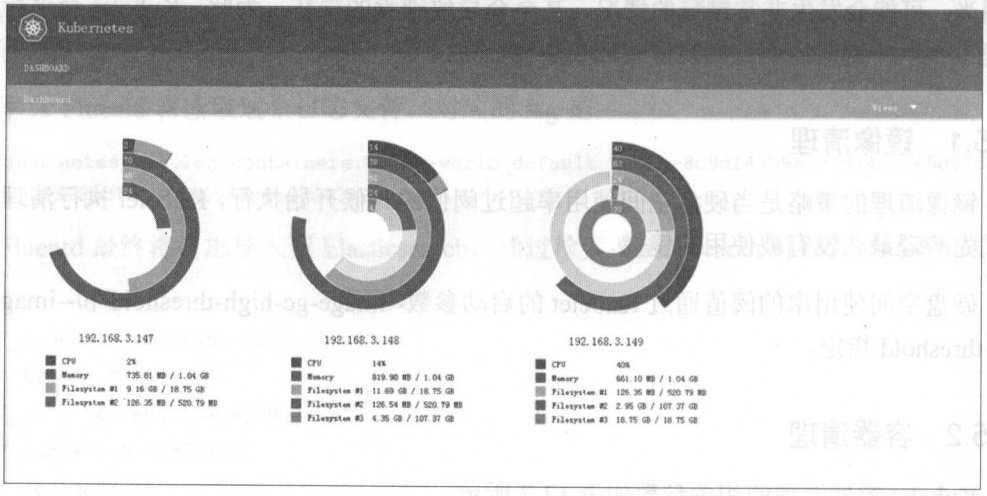


图 12-4　Kube UI 首页

单击首页右上角的 View 按钮，包含 Explore、Pods、Nodes、Replication Controllers、Services 和 Events 等选项。单击进入 Explore 页面，会列出所有 Pod、Replication Controller 和 Service，支持筛选和排列展示，如图 12-5 所示。

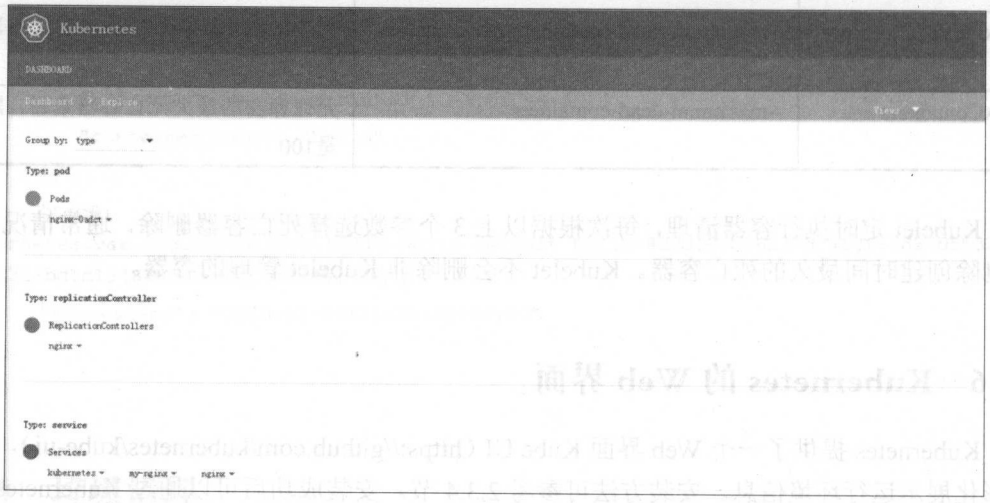


图 12-5　Explore 页面

同时可以分别查询 Pod、Replication Controller 和 Service 的详细信息，比如查询 Service 的详细信息，如图 12-6 所示。

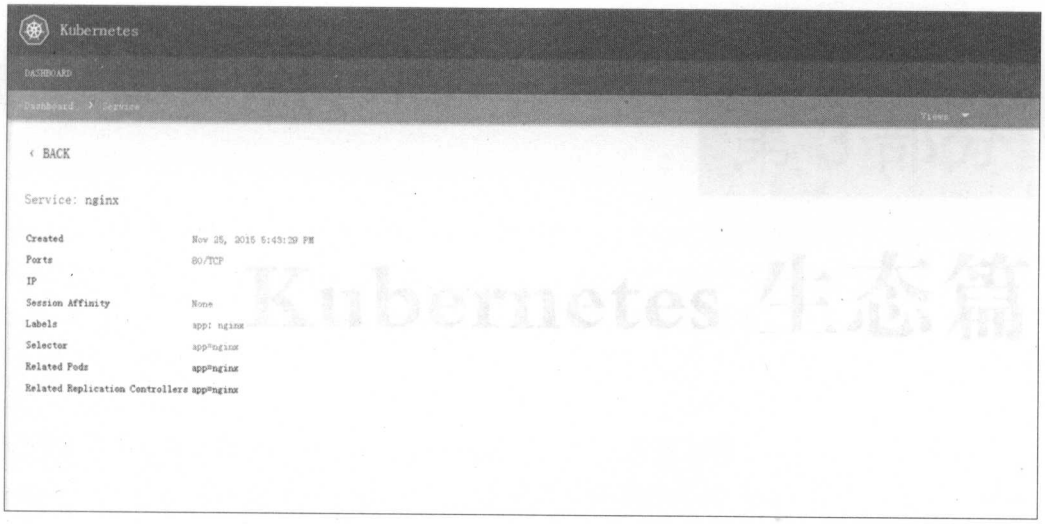


图 12-6　查询 Service 的详细信息

其他页面也列出了相应的信息，比如查询 Event，如图 12-7 所示。

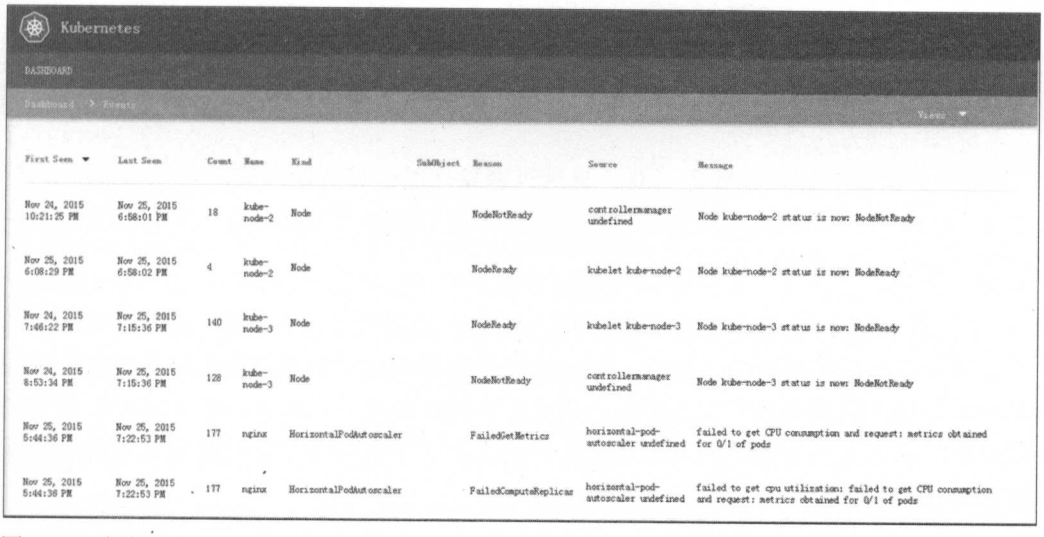
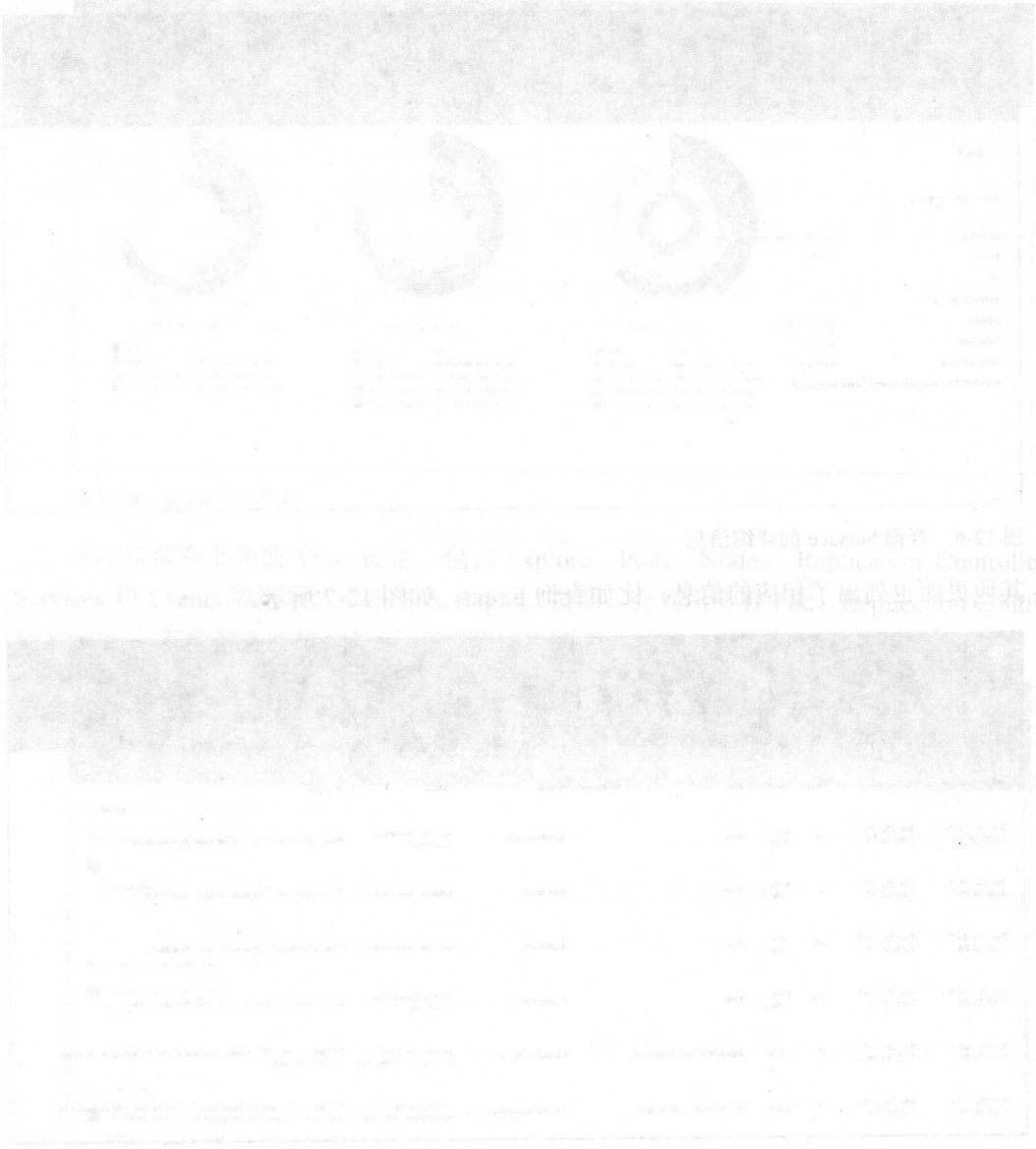


图 12-7　查询 Event

Kube UI 只能简单地查看一些信息，不能进行修改，新的 Kubernetes Dashboard（https://github.com/kubernetes/dashboard）正在开发中，将支持功能更加强大的 Web 界面。



第 3 部分

Kubernetes 生态篇

第 13 章

CoreOS

CoreOS 是容器生态圈重要的一环，与 Kubernetes 和 Docker 都有着非常密切的关系。本章将简单介绍 CoreOS，然后讲解如何安装 CoreOS，以及如何使用 CoreOS 运行 Kubernetes。

13.1 CoreOS 介绍

CoreOS 是一个轻量级、容器化的 Linux 发行版，借助了以 Docker 为代表的容器技术，专为大型数据中心而设计，旨在通过轻量的系统架构和灵活的应用程序部署能力简化数据中心的维护成本和复杂度。

CoreOS 没有提供包管理工具，是通过容器化的运算环境向应用程序提供运算资源的。在 CoreOS 中，所有应用程序都被装在容器中运行在操作系统之上，可以很轻松地将应用程序在操作系统之间转移。

应用程序之间共享系统内核和资源，但是彼此之间又互不可见。这意味着应用程序将不会再被直接安装到操作系统中，而是运行在容器中。这种方式使得操作系统、应用程序及运行环境之间的耦合度大大降低。相对于传统的部署方式而言，在 CoreOS 集群中部署应用程序更加灵活便捷，应用程序运行环境之间的干扰更少，而且操作系统自身的维护也更加容易。

借助容器的能力，CoreOS 可以剔除任何不必要的软件和服务，最小化定制化 Linux 系统。因此很小很轻量，管理员操心的事情会少很多，允许快速修复，占据的空间也很小。

在一定程度上减轻了维护一个服务器集群的复杂度，可帮助用户从烦琐的系统及软件维护工作中解脱出来。

13.2 CoreOS 工具链

13.2.1 Etcd

在 CoreOS 集群中处于骨架地位的是 Etcd，Etcd 是 Core 团队开发的一个高可用的键值存储系统，灵感来自于 ZooKeeper 和 Doozer。Etcd 以默认的形式安装于每个 CoreOS 系统之中，CoreOS 集群中的程序和服务可以通过 Etcd 共享信息或服务发现。

13.2.2 Flannel

Flannel 是 CoreOS 团队开发的覆盖网络工具，CoreOS 集群使用 Flannel 创建的一个容器覆盖网络，实现容器之间的通信报文，实现跨主机通信。

13.2.3 Rocket

Rocket 是 CoreOS 推出的一款容器引擎，和 Docker 类似，帮助开发者打包应用和依赖包到可移植容器中，简化搭建环境等部署工作。Rocket 同 Docker 相比更加专注于容器核心技术和容器标准。

13.2.4 Systemd

Systemd 是 Linux 下的一种 Init 软件，由 Lennart Poettering 带头开发，并在 LGPL 2.1 及其后续版本许可证下开源发布框架以表示系统服务间的依赖关系，并依此实现系统初始化时服务的并行启动，同时达到降低 Shell 的系统开销的效果，最终代替现在常用的 System V 与 BSD 风格的 Init 程序，CoreOS 已将 Systemd 作为 Linux 发行版的 Init 系统。

13.2.5 Fleet

Fleet 是一个通过 Systemd 对 CoreOS 集群进行控制和管理的工具。Fleet 与 Systemd 之间通过 D-Bus API 进行交互，每个 Fleet Agent 之间通过 Etcd 服务来注册和同步数据。Fleet 提供的功能非常丰富，包括查看集群中服务器的状态、启动或终止 Docker 容器、读取日

志内容等。更为重要的是，Fleet 可以确保集群中的服务一直处于可用状态。当出现某个通过 Fleet 创建的服务在集群中不可用时，如由于某台主机因为硬件或网络故障从集群中脱离时，原本运行在这台服务器中的一系列服务将通过 Fleet 被重新分配到其他可用服务器中。虽然当前 Fleet 还处于初期状态，但是其管理 CoreOS 集群的能力是非常有效的，并且仍然有很大的扩展空间，目前已提供简单的 API 接口供用户集成。

13.3 CoreOS 实践

13.3.1 安装 CoreOS

本节将使用 Vagrant 部署 CoreOS，这可以说是最简单的方式了，可以使用本地虚拟机快速部署 CoreOS。

准备工作

在机器上预先装好 VirtualBox 和 Vagrant：

- VirtualBox 4.3.10 以上版本
- Vagrant 1.6 以上版本

下载 CoreOS-Vagrant 仓库，其中包含使用 Vagrant 部署 CoreOS 的配置文件：

```
$ git clone https://github.com/coreos/coreos-vagrant
$ cd coreos-vagrant
```

配置

在 CoreOS-Vagrant 仓库目录下，config.rb.sample 和 user-data.sample 是两个模板文件，根据这两个模板文件可以对 CoreOS 部署进行自定义配置，通过模板文件生成配置文件：

```
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
```

配置文件 user-data 将作为 CoreOS 操作系统初始化时使用的 Cloud-Init 配置文件，Cloud-Init 是专为云环境中虚拟机而开发的工具，它根据配置文件对虚拟机进行系统初始化配置，配置文件使用 YAML 文件格式，并且需要包含#cloud-config：

```
#cloud-config
```

```
coreos:
  etcd2:
    #generate a new token for each unique cluster from https://discovery.etcd.io/new
    #discovery: https://discovery.etcd.io/<token>
    # multi-region and multi-cloud deployments need to use $public_ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{ "Network":
"10.1.0.0/16" }'
      command: start
    - name: docker-tcp.socket
      command: start
      enable: true
      content: |
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        Service=docker.service
        BindIPv6Only=both
```

```
[Install]
WantedBy=sockets.target
```

Cloud-Init 配置文件中包含了 Etcd、Fleet、Flannel 和 Docker 的服务启动参数，其中使用两个变量$private_ipv4 和$public_ipv4，它们会在实际运行的时候被自动替换为虚拟机的真实外网 IP 和内网 IP 地址。用户可以根据需要，在配置中添加更多定制化的服务和配置，具体信息可参考 https://coreos.com/docs/cluster-management/setup/cloudinit-cloud-config。

另外，CoreOS 集群是用 Etcd 进行服务发现的，Cloud-Init 配置文件中的.coreos.etcd2.discovery 就是用来配置一个外部 Etcd 服务的 URL，CoreOS 启动时通过这个集群标识 URL 地址自动进入同一个集群中，这就实现了无须人工干预的集群服务器自发现。我们可以通过 Etcd 发现服务（discovery.etcd.io）申请创建 URL，如果没有配置的话，在 config.rb 中也会自动申请创建进行配置。

配置文件 config.rb 中包含了 Vagrant 虚拟机的配置，通过这个文件可以覆盖 Vagrantfile 里的参数。我们主要关注$num_instances 和$update_channel 这两个参数：

- $num_instances 表示将启动的 CoreOS 集群中需要包含主机实例的数量。
- $update_channel 表示启动的 CoreOS 实例使用的升级通道，可以是 stable、eta 或 alpha。

现在配置 config.rb，将部署有 1 个节点的 CoreOS 集群，并使用 alpha 升级通道：

```
$num_instances=1
$update_channel='alpha'
...
```

部署

使用 Vagrant 部署 CoreOS：

```
$ vagrant up
```

部署成功后，可以查询虚拟机的状态：

```
$ vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   running (virtualbox)
```

```
This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

在 CoreOS 集群运行后，集群的实例之间通过 Etcd 实现自发现服务，同时运行起 Docker 和 Flannel，形成连通的容器集群，可以 SSH 到 CoreOS 节点中：

```
$ vagrant ssh core-01
```

然后在 CoreOS 节点中查询服务状态：

```
$ sudo systemctl status etcd2 docker flanneld
```

13.3.2 使用 CoreOS 运行 Kubernetes

Kubernetes 是一个容器集群管理平台，而 CoreOS 是基于容器的集群操作系统，两者可以说有着千丝万缕的关系。CoreOS 集群已经原生运行 Docker，同时使用 Flannel 搭建出容器的覆盖网络，因此 Kubernetes 非常适合运行在 CoreOS 之上。

现在运行 3 个节点的 CoreOS 集群，我们将在 CoreOS 集群之上运行 Kubernetes，其中所有节点将作为 Kubernetes Node，而节点 core-01 同时作为 Kubernetes Master。

启动 Kubelet

CoreOS 在 alpha 开发版（773.1.0 以后版本）中集成了 Kubernetes 的核心组件 Kubelet，我们通过在所有节点中进行简单配置就可以启动 kubelet，从而作为 Kubernetes Node。

创建 kubelet Systemd 单元文件/etc/systemd/system/kubelet.service：

```
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /etc/kubernetes/manifests
ExecStart=/usr/bin/kubelet \
--api-servers=http://kuber-apiserver:8080 \
--allow-privileged=true \
--config=/etc/kubernetes/manifests \
--v=2
```

```
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target :qw
```

其中，我们将在节点 core-01 上运行 Kubernetes API Server，所以--api-servers 设置的 URL 指向 core-01。

配置好 Systemd 单元文件后，使用 systemctl 命令启动 Kubelet：

```
$ sudo systemctl daemon-reload
$ sudo systemctl start kubelet
```

运行后用 systemctl 命令查询 Kubelet 是否运行：

```
$ sudo systemctl status kubelet
```

另外设置 kubelet 为开机自启动：

```
$ sudo systemctl enable kubelet
```

部署 Kubernetes Master

现在在节点 core-01 上运行 Kubernetes Master，首先下载 Kubernetes Master 的 Pod 定义文件：

```
$ wget https://raw.githubusercontent.com/coreos/pods/master/kubernetes.yaml
```

然后将其放入 Kubelet 的 Manifest 目录，由 Kubelet 以 Daemon Pod 的形式运行 Kubernetes Master 各组件：

```
$ sudo cp kubernetes.yaml /etc/kubernetes/manifests/
```

确保镜像下载正常，那么一段时间后，Kubernetes Master 的各个组件将运行起来。

第 14 章

Etcd

Kubernetes 使用 Etcd 进行存储，理解和掌握 Etcd 对于管理 Kubernetes 至关重要。本章将介绍 Etcd，包括基本概念和结构组织，然后详细说明如何运行和管理 Etcd。

14.1 Etcd 介绍

Etcd 是一个高可用的键值存储系统，Etcd 的灵感来自于 ZooKeeper 和 Doozer，通过 Raft 共识算法（The Raft Consensus Algorithm）处理日志复制以保证强一致性。

Etcd 中的主要概念如下所示。

- Raft：Etcd 所采用的保证分布式系统强一致性的算法。
- Node：一个 Raft 状态机实例。
- Member：一个 Etcd 实例，它管理着一个 Node，并且可以为客户端请求提供服务。
- Cluster：由多个 Member 构成可以协同工作的 Etcd 集群。
- Peer：对同一个 Etcd 集群中另外一个 Member 的称呼。
- Client：向 Etcd 集群发送 HTTP 请求的客户端。
- WAL：预写式日志，Etcd 用于持久化存储的日志格式。
- Snapshot：Etcd 防止 WAL 文件过多而设置的快照，存储 Etcd 数据状态。
- Proxy：Etcd 的一种模式，为 Etcd 集群提供反向代理服务。
- Leader：Raft 算法中通过竞选而产生的处理所有数据提交的节点。

- Follower：竞选失败的节点作为 Raft 中的从属节点，为算法提供强一致性保证。
- Candidate：当 Follower 超过一定时间接收不到 Leader 的心跳时转变为 Candidate 开始竞选。
- Term：某个节点成为 Leader 到下一次竞选的时间，称为一个 Term。
- Index：数据项编号。Raft 中通过 Term 和 Index 来定位数据。

14.2 Etcd 的结构

Etcd 主要分为 4 个部分，如图 14-1 所示。

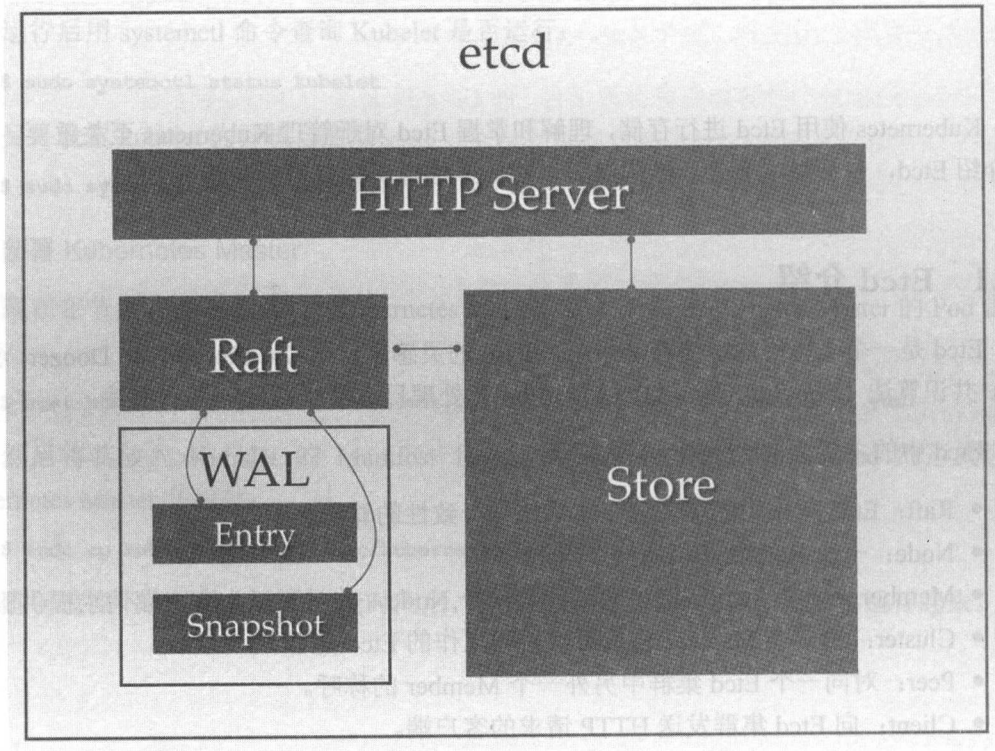


图 14-1 Etcd 的结构

- HTTP Server：用于处理用户发送的 API 请求以及其他 Etcd 节点的同步与心跳信息请求。

- Store：用于处理 Etcd 支持的各类功能的事务，包括数据索引、节点状态变更、监控与反馈、事件处理与执行等，是 Etcd 对用户提供的大多数 API 功能的具体实现。
- Raft：强一致性算法的具体实现，是 Etcd 的核心。
- WAL：Write Ahead Log（预写式日志），是 Etcd 的数据存储方式，是用于向系统提供原子性和持久性的一系列技术。在使用 WAL 的时候，所有的修改在提交之前都要先写入日志文件中。Etcd 的 WAL 由日志存储与快照存储两部分组成，其中，Entry 负责存储具体日志的内容，而 Snapshot 负责在日志内容发生变化的时候保存 Raft 的状态。WAL 会在本地磁盘的一个指定目录下分别存放日志条目与快照内容。

通常，一个用户的请求发送过来，会经由 HTTP Server 转发给 Store 进行具体的事务处理。如果涉及节点的修改，则交给 Raft 模块进行状态的变更、日志的记录，然后再同步给其他 Etcd 节点以确认数据提交，最后进行数据的提交、再次同步。

Etcd 属于分布式架构，其中的通信模型有两种，一种是 Etcd Client 同 Etcd Server 之间的通信，另一种是 Etcd Peer 之间的通信。

14.2.1 Client-to-Server

在默认设置下，Etcd 通过主机的 2379/4001 端口向 Client 提供服务，每个主机上的应用程序都可以通过主机的 2379/4001 端口以 HTTP + JSON 的方式向 Etcd 读写数据。写入的数据会由 Etcd 同步到集群的其他节点中，如图 14-2 所示。

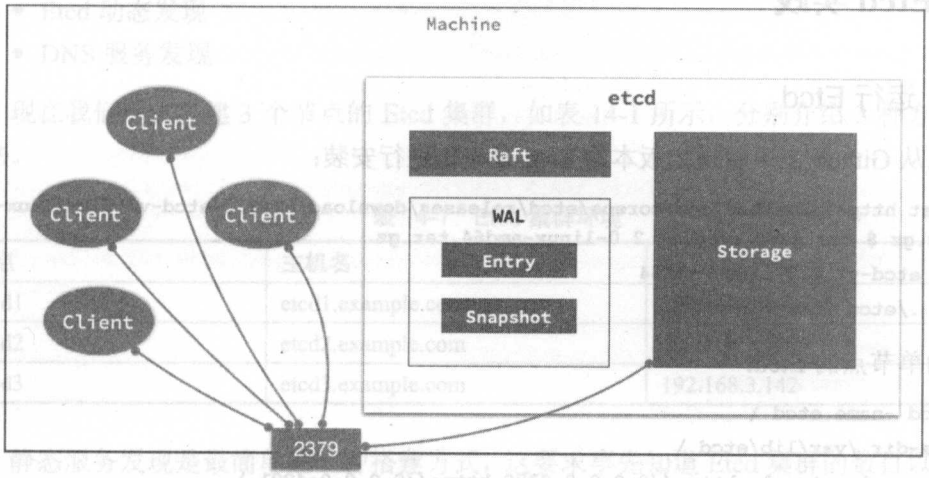


图 14-2 Etcd Client 同 Etcd Server 之间的通信

14.2.2 Peer-to-Peer

在默认设置下，Etcd 通过主机的 2380/7001 端口在各个节点中同步 Raft 状态及数据，如图 14-3 所示。

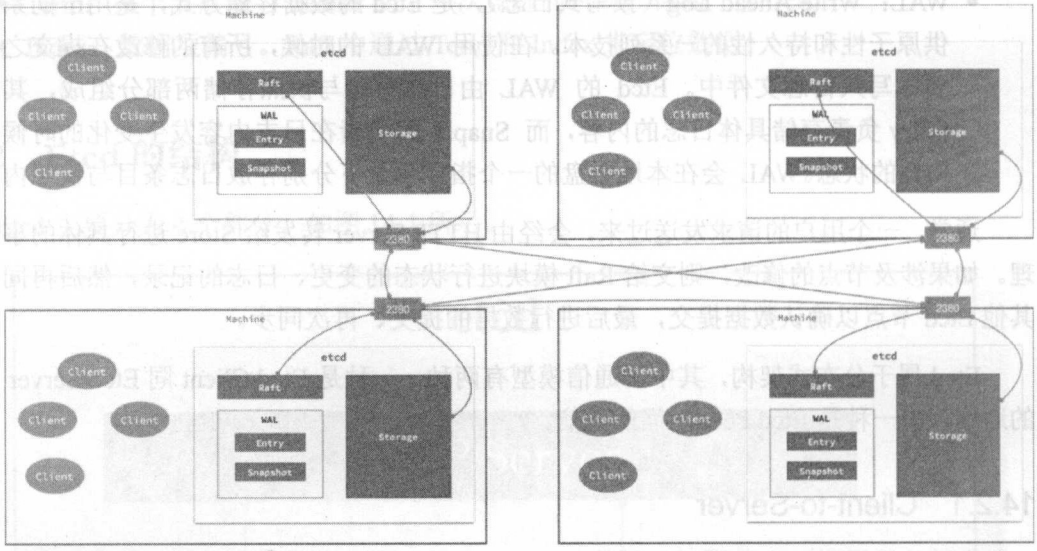


图 14-3 Etcd Peer 之间的通信

14.3 Etcd 实践

14.3.1 运行 Etcd

可以从 Github 上下载指定版本的 Etcd 发布包进行安装：

```
$ wget https://github.com/coreos/etcd/releases/download/v2.2.0/etcd-v2.2.0-linux-amd64.tar.gz $ tar xzvf etcd-v2.2.0-linux-amd64.tar.gz
$ cd etcd-v2.2.0-linux-amd64
$ cp ./etcd /usr/bin
```

启动单节点的 Etcd：

```
$ etcd -name etcd \
-data-dir /var/lib/etcd \
-listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
-advertise-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001
```

在 Etcd 的启动参数中，-name 设置 Etcd 节点的名称，-data-dir 设置 Etcd 的数据目录。另外，-listen-client-urls 设置了 Etcd 向 Client 提供服务的监听 URL，多个 URL 直接用逗号分隔，而-advertise-client-urls 指定了 Etcd 对外广播的 URL，需要在-listen-client-urls 配置 URL 范围。

14.3.2 Etcd 集群化

Etcd 集群的工作原理基于 Raft 共识算法。Raft 共识算法的优点在于，可以在高效地解决分布式系统中各个节点日志内容一致性问题的同时，也使得集群具备一定的容错能力，即使集群中出现部分节点故障、网络故障等问题，仍可保证其余大多数节点正确地步进，甚至当更多的节点（一般来说超过集群节点总数的一半）出现故障而导致集群不可用时，依然可以保证节点中的数据不会出现错误的结果。

因为需要选举 Leader 节点，所以 Etcd 集群至少需要两个节点，但是如果是两个成员，那么在一台无法正常运转的情况下，剩下的一个也无法正常工作，一般建议是 3~9 个节点，并且一个重要的集群优化策略是要保障集群中活跃节点的数目始终为奇数个。

14.3.2.1 搭建集群

要搭建 Etcd 集群，需要让 Etcd 节点互相发现，目前提供的发现方式有 3 种：

- 静态发现
- Etcd 动态发现
- DNS 服务发现

现在我们通过搭建 3 个节点的 Etcd 集群，如表 14-1 所示，分别介绍 3 种方式的配置方法。

表 14-1 Etcd 集群环境

节点	主机名	IP
Etcd1	etcd1.example.com	192.168.3.140
Etcd2	etcd2.example.com	192.168.3.141
Etcd3	etcd3.example.com	192.168.3.142

静态服务发现是最简单的一种搭建方式，这要求事先知道 Etcd 集群的数目以及每个节点的地址，然后在每个 Etcd 节点静态地配置初始的 Etcd 集群信息：

```
    -initial-cluster etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,etcd3=
http://192.168.3.142:2380
    -initial-cluster-state new
```

其中，-initial-cluster-state=new 表示这是在从无到有搭建 Etcd 集群。参数-initial-cluster 描述了这个新集群中共有哪些节点，其中每个节点用 name=url 的形式描述，节点之间用逗号分隔，并且指定的 URL 是 Etcd 的广播 Peer 地址，即需要和--initial-advertise-peer-urls 参数设置得一致。

分别在 3 个节点上启动 Etcd：

- Etcd1

```
$ etcd -name etcd1 \
-data-dir /var/lib/etcd1 \
-listen-peer-urls http://192.168.3.140:2380 \
-initial-advertise-peer-urls http://192.168.3.140:2380 \
-listen-client-urls http://192.168.3.140:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.140:2379 \
-initial-cluster
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,etcd3=http://192.168.3.14
2:2380 \
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \
-data-dir /var/lib/etcd2 \
-listen-peer-urls http://192.168.3.141:2380 \
-initial-advertise-peer-urls http://192.168.3.141:2380 \
-listen-client-urls http://192.168.3.141:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.141:2379 \
-initial-cluster etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,etcd3=
http://192.168.3.142:2380 \
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \
-data-dir /var/lib/etcd3 \
-listen-peer-urls http://192.168.3.142:2380 \
-initial-advertise-peer-urls http://192.168.3.142:2380 \
-listen-client-urls http://192.168.3.142:2379,http://127.0.0.1:2379 \
```

```
    -advertise-client-urls http://192.168.3.142:2379 \
    -initial-cluster etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,etcd3=
http://192.168.3.142:2380 \
    -initial-cluster-state new
```

动态发现适用于无法事先知道 Etcd 集群规模和每个节点地址的场景，包括 Etcd 动态发现和 DNS 动态发现。

Etcd 动态发现

Etcd 动态发现方式是利用一个已有的 Etcd 服务来提供服务发现，从而搭建出一个新的 Etcd 集群。

CoreOS 官方提供了免费的 Etcd 发现服务（discovery.etcd.io），可以用来帮助初始化新 Etcd 集群。通过浏览器或者命令行 curl 访问地址 https://discovery.etcd.io，可以得到一个新的集群标识 URL，比如创建规模数目为 3 的集群标识 URL：

```
$ curl -w "\n" 'https://discovery.etcd.io/new?size=3'
https://discovery.etcd.io/e1839e0d76aa687cb7c9bafcdf420f66
```

输出的 URL 是搭建 Etcd 集群需要使用到的，可通过环境变量设置：

```
$ export ETCD_CLUSTER_DISCOVERY_URL=https://discovery.etcd.io/
e1839e0d76aa687cb7c9bafcdf420f66
```

如果无法使用 discovery.etcd.io，也可以利用已有的 Etcd 服务，比如 Etcd 服务访问地址是 http://myetcd.local，我们同样需要创建新的集群标识 URL，首先生成一个 UUID：

```
$ export ECTD_CLUSTER_UUID=$(cat /dev/urandom | base64 | tr -d "=+/" | dd bs=40 count=1
2> /dev/null)
$ echo $ECTD_CLUSTER_UUID
Batno20LUGhFICsmuZiQgM7tGlZ84h2jLnmkLyf2
```

然后创建集群标识 Key，其中设置集群规模数目为 3：

```
$ curl -X PUT http://myetcd.local/v2/keys/discovery/${ECTD_CLUSTER_UUID}/_config/size
-d value=3
{"action":"set","node":{"key":"/discovery/Batno20LUGhFICsmuZiQgM7tGlZ84h2jLnmkLyf2/_
config/size","value":"3","modifiedIndex":3998,"createdIndex":3998}}
```

通过环境变量设置集群标识 URL：

```
$ export ETCD_CLUSTER_DISCOVERY_URL=http://myetcd.local/v2/keys/discovery/${ECTD_
CLUSTER_UUID}
```

在搭建新集群的时候，需要通过-discovery 参数指定我们创建的 URL，分别在 3 个节点上启动 Etcd：

- Etcd1

```
$ etcd -name etcd1 \
-data-dir /var/lib/etcd1 \
-listen-peer-urls http://192.168.3.140:2380 \
-initial-advertise-peer-urls http://192.168.3.140:2380 \
-listen-client-urls http://192.168.3.140:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.140:2379 \
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \
-data-dir /var/lib/etcd2 \
-listen-peer-urls http://192.168.3.141:2380 \
-initial-advertise-peer-urls http://192.168.3.141:2380 \
-listen-client-urls http://192.168.3.141:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.141:2379 \
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \
-data-dir /var/lib/etcd3 \
-listen-peer-urls http://192.168.3.142:2380 \
-initial-advertise-peer-urls http://192.168.3.142:2380 \
-listen-client-urls http://192.168.3.142:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.142:2379 \
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \
-initial-cluster-state new
```

DNS 动态发现

在 DNS 中，一个域名能够关联若干种资源记录，除了我们比较熟悉的 A 记录（用于地址查询）或者是 MX 记录（用于邮件服务查询），DNS 还定义了 SRV 记录，可以用于服务发现。

Etcd 支持利用 DNS SRV 记录实现互相发现，通过-discovery-srv 参数设置 DNS SRV 域名，比如设置成 example.com，然后 Etcd 会依次进行查询。

```
_etcd-server-ssl._tcp.example.com
_etcd-server._tcp.example.com
```

如果_etcd-server-ssl._tcp.example.com 解析成功，那么 Etcd 就会使用 HTTPS/SSL 启动。

对于我们要搭建的集群，首先要创建 DNS SRV 记录：

```
$ dig +noall +answer SRV _etcd-server._tcp.example.com
_etcd-server._tcp.example.com. 300 IN   SRV 0 0 2380 etcd1.example.com.
_etcd-server._tcp.example.com. 300 IN   SRV 0 0 2380 etcd2.example.com.
_etcd-server._tcp.example.com. 300 IN   SRV 0 0 2380 etcd3.example.com.

$ dig +noall +answer etcd1.example.com etcd2.example.com etcd3.example.com
etcd1.example.com. 300 IN A   192.168.3.140
etcd2.example.com. 300 IN A   192.168.3.141
etcd3.example.com. 300 IN A   192.168.3.142
```

然后分别在 3 个节点上启动 Etcd。

- Etcd1

```
$ etcd -name etcd1 \
-data-dir /var/lib/etcd1 \
-listen-client-urls http://etcd1.example.com:2379 \
-advertise-client-urls http://etcd1.example.com:2379 \
-listen-peer-urls http://etcd1.example.com:2380 \
-initial-advertise-peer-urls http://etcd1.example.com:2380 \
-discovery-srv example.com \
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \
-data-dir /var/lib/etcd2 \
-listen-client-urls http://etcd2.example.com:2379 \
-advertise-client-urls http://etcd2.example.com:2379 \
-listen-peer-urls http://etcd2.example.com:2380 \
-initial-advertise-peer-urls http://etcd2.example.com:2380 \
-discovery-srv example.com \
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \
-data-dir /var/lib/etcd3 \
-listen-client-urls http://etcd3.example.com:2379 \
-advertise-client-urls http://etcd3.example.com:2379 \
-listen-peer-urls http://etcd3.example.com:2380 \
-initial-advertise-peer-urls http://etcd3.example.com:2380 \
-discovery-srv example.com \
-initial-cluster-state new
```

14.3.2.2 集群管理

Etcd 集群搭建完成后，可以查询集群成员：

```
$ etcdctl member list
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380 clientURLs=http://192.168.3.140:2379
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380 clientURLs=http://192.168.3.141:2379
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380 clientURLs=http://192.168.3.142:2379
```

以及集群的健康状态：

```
$ etcdctl cluster-health
member f042ea167d8f2828 is healthy: got healthy result from http://192.168.3.140:2379
member 156ce626171618a6 is healthy: got healthy result from http://192.168.3.141:2379
member c99b1511c3ade2bb is healthy: got healthy result from http://192.168.3.142:2379
cluster is healthy
```

增加成员

添加成员信息：

```
$ etcdctl member add etcd4 http://192.168.3.143:2380
added member 4d906f7a642c3f23 to cluster

ETCD_NAME="etcd4"
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,etcd3=http://192.168.3.142:2380, etcd4=http://192.168.3.143:2380"
ETCD_INITIAL_CLUSTER_STATE=existing
```

添加成功后返回的信息中提示设置环境变量 etcd_name、etcd_initial_cluster 和 etcd_initial_cluster_state，这将可以在启动新的 Etcd 成员时使用：

```
$ etcd -name etcd4 \
-data-dir /var/lib/etcd4 \
-listen-client-urls http://192.168.3.143:2379 \
-advertise-client-urls http://192.168.3.143:2379 \
-listen-peer-urls http://192.168.3.143:2380 \
-initial-advertise-peer-urls http://192.168.3.143:2380 \
-initial-cluster $ETCD_INITIAL_CLUSTER \
-initial-cluster-state existing
```

最后，可以查询到新的成员列表：

```
$ etcdctl member list
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380 clientURLs=http://192.168.3.140:2379
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380 clientURLs=http://192.168.3.141:2379
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380 clientURLs=http://192.168.3.142:2379
4d906f7a642c3f23: name=etcd4 peerURLs=http://192.168.3.143:2380 clientURLs=http://192.168.3.143:2379
```

删除成员

```
$ etcdctl member remove 4d906f7a642c3f23
Removed member 4d906f7a642c3f23 from cluster

$ etcdctl  member list
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380 clientURLs=http://192.168.3.140:2379
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380 clientURLs=http://192.168.3.141:2379
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380 clientURLs=http://192.168.3.142:2379
```

更新成员

```
$ etcdctl member update f042ea167d8f2828 http://etcd1.example.com:2380
Updated member with ID f042ea167d8f2828 in cluster

$ etcdctl  member list
```

```
    f042ea167d8f2828: name=etcd1 peerURLs=http://etcd1.example.com:2380 clientURLs=http://
192.168.3.140:2379
    156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380 clientURLs=http://
192.168.3.141:2379
    c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380 clientURLs=http://
192.168.3.142:2379
```

迁移成员

当需要迁移成员的时候，实际上等效于删除一个旧成员，再新增一个新成员，让数据自动切换，但是如果数据过大（大于 50MB），会影响集群的状态。更安全的做法是人为地迁移数据，比如现在需要将 Etcd 成员 etcd3 从机器 1（192.168.3.142）迁移机器 2（192.168.3.143），步骤如下。

- 首先在机器 1 上停止 Etcd 进程，打包数据并复制到机器 2：

```
$ kill `pgrep etcd`
$ tar -cvzf etcd3.data.tar.gz /var/lib/etcd3
$ scp etcd3.data.tar.gz 192.168.3.143:~/
```

紧接着更新 Etcd 成员：

```
$ etcdctl member update c99b1511c3ade2bb http://192.168.3.143:2380
Updated member with ID c99b1511c3ade2bb in cluster
```

然后在机器 2 上启动 Etcd：

```
$ tar -xzvf etcd3.data.tar.gz -C /var/lib/etcd3
$ etcd -name etcd3 \
-data-dir /var/lib/etcd3 \
-listen-peer-urls http://192.168.3.143:2380 \
-initial-advertise-peer-urls http://192.168.3.143:2380 \
-listen-client-urls http://192.168.3.143:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://192.168.3.143:2379
```

14.3.3 Etcd Proxy 模式

Etcd 可以设置为 Proxy 模式，此时 Etcd Proxy 并不是直接加入到数据强一致性的 Etcd 集群中，所以 Etcd Proxy 并没有增加集群的可靠性，当然也没有降低集群的写入性能。Etcd Proxy 只是作为一个反向代理把客户的请求转发给可用的 Etcd 集群。一个典型的应用场景是，在客户端机器本地运行 Etcd Proxy 来对接真正的 Etcd 服务端集群，对客户端应用来说

可以直接访问本地的 Etcd Proxy（http://127.0.0.1:2379），而不用感知 Etcd 服务端集群的变化，因为 Etcd Proxy 会维护同 Etcd 服务端集群的同步。

启动一个 Etcd Proxy，对接 Etcd 集群：

- 静态发现

```
$ etcd -proxy on \
-listen-client-urls http://127.0.0.1:4001 \
-initial-cluster etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,
etcd3=http://192.168.3.142:2380
```

- Etcd 动态发现

```
$ etcd -proxy on \
-listen-client-urls http://127.0.0.1:4001 \
-discovery ${DISCOVERY_URL}
```

- DNS 动态发现

```
$ etcd --proxy on \
-listen-client-urls http://127.0.0.1:4001 \
-discovery-srv example.com
```

14.3.4 Etcd 的安全模式

Etcd 中的通信包括 Client-to-Server 和 Peer-to-Peer，支持 SSL/TLS 加密和 Client Certificate Authentication 认证。

Client-to-Server 通信

- 开启 HTTPS

开启 HTTPS，需要 CA 证书（server-ca.crt）和该 CA 签发的服务端密钥对（server.crt/server.key）。运行 Etcd：

```
$ etcd -name etcd1 \
-cert-file=/path/to/server.crt -key-file=/path/to/server.key \
-advertise-client-urls=https://127.0.0.1:2379 \
-listen-client-urls=https://127.0.0.1:2379
```

Etcd Client 通过 HTTPS 访问 Etcd Server 时就需要提供 CA 证书（server-ca.crt）：

```
$ etcdctl --ca-file=/path/to/server-ca.crt -C https://127.0.0.1:2379 cluster-health
```

```
member ce2a822cea30bfca is healthy: got healthy result from https://127.0.0.1:2379
cluster is healthy
```

- 开启 Client Certificate Authentication

在开启 HTTPS 后，可以开启 Client Certificate Authentication。这样一来，Etcd Client 在访问 Etcd Server 的时候就需要提供 Client Certificate 进行认证，认证成功才能有权限访问。

同样的，需要 CA 证书（client-ca.crt）和该 CA 签发的客户端密钥对（client.crt/client.key），签发客户端密钥对和签发服务端的 CA 可以一样，这实际上并不冲突，因为一个 CA 可以同时签发服务端密钥对和客户端密钥对。运行 Etcd：

```
$ etcd -name etcd1 -data-dir etcd1 \
-client-cert-auth -trusted-ca-file=/path/to/client-ca.crt \
-cert-file=/path/to/server.crt -key-file=/path/to/server.key \
-advertise-client-urls https://127.0.0.1:2379 \
-listen-client-urls https://127.0.0.1:2379
```

如 Etcd Client 按照之前的命令访问 Etcd Server，会报错：

```
$ etcdctl --ca-file=/path/to/server-ca.crt -C https://127.0.0.1:2379 cluster-health
cluster may be unhealthy: failed to list members
Error: client: etcd cluster is unavailable or misconfigured
error #0: remote error: bad certificate
```

因为 Etcd Client 需要提供 Client Certificate，才能认证成功：

```
$ etcdctl --ca-file=/path/to/server-ca.crt \
--key-file=/path/to/client.key --cert-file=/path/to/client.crt \
-C https://127.0.0.1:2379 \
cluster-health
member ce2a822cea30bfca is healthy: got healthy result from https://127.0.0.1:2379
cluster is healthy
```

Peer-to-Peer 通信

- 开启 HTTPS 和 Client Certificate Authentication

类似的，Etcd Peer 之间的通信开启 HTTPS 和 Client Certificate Authentication，需要准备 CA 证书（ca.crt），为每个 Peer 签发一个密钥对（server1.key 和 server1.crt，server2.key 和 server2.crt）。

- Etcd1

```
$ etcd -name etcd1 \
-peer-client-cert-auth -peer-trusted-ca-file=/path/to/ca.crt \
-peer-cert-file=/path/to/server1.crt -peer-key-file=/path/to/server1.key \
-listen-peer-urls=https://192.168.3.140:2380 \
-initial-advertise-peer-urls=https://192.168.3.140:2380 \
-initial-cluster etcd1=https://192.168.3.140:2380,etcd2=https://192.168.3.141:2380 \
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \
-peer-client-cert-auth -peer-trusted-ca-file=/path/to/ca.crt \
-peer-cert-file=/path/to/server2.crt -peer-key-file=/path/to/server2.key \
-listen-peer-urls=https://192.168.3.141:2380 \
-initial-advertise-peer-urls=https://192.168.3.141:2380 \
-initial-cluster etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380,
etcd3=http://192.168.3.142:2380 \
-initial-cluster etcd1=https://192.168.3.140:2380,etcd2=https://192.168.3.141:2380 \
-initial-cluster-state new
```

第 15 章

Mesos

Mesos 是一个成熟的架构，一方面提供了同 Kubernetes 类似的容器集群管理方案；另一方面，Mesos 正积极同 Kubernetes 进行整合，Mesos 和 Kubernetes 亦敌亦友。本章将介绍 Mesos，以及 Marathon 和 K8SM，最后说明如何使用这 3 个项目。

15.1 Mesos 介绍

Apache Mesos 是由加州大学伯克利分校的 AMPLab 首先开发的一款开源集群管理软件，支持 Hadoop、Elasticsearch、Spark、Storm 和 Kafka 等架构。其开源性越来越受到一些大型云计算公司的青睐，Twitter 和 Airbnb 公司已经将其大规模应用在其数据中心中了。现在，一家初创公司 Mesosphere 将 Mesos 定位为数据中心操作系统（Data Center Operating System，DCOS），使之步入主流。

Mesos 在多种不同类型的工作之间共享机器（或者节点）的可用资源，如图 15-1 所示。Mesos 可以看作是数据中心的内核，提供所有节点资源的统一视图，所起的作用类似于操作系统内核在单台机器上的作用，可以无缝地访问多节点资源。

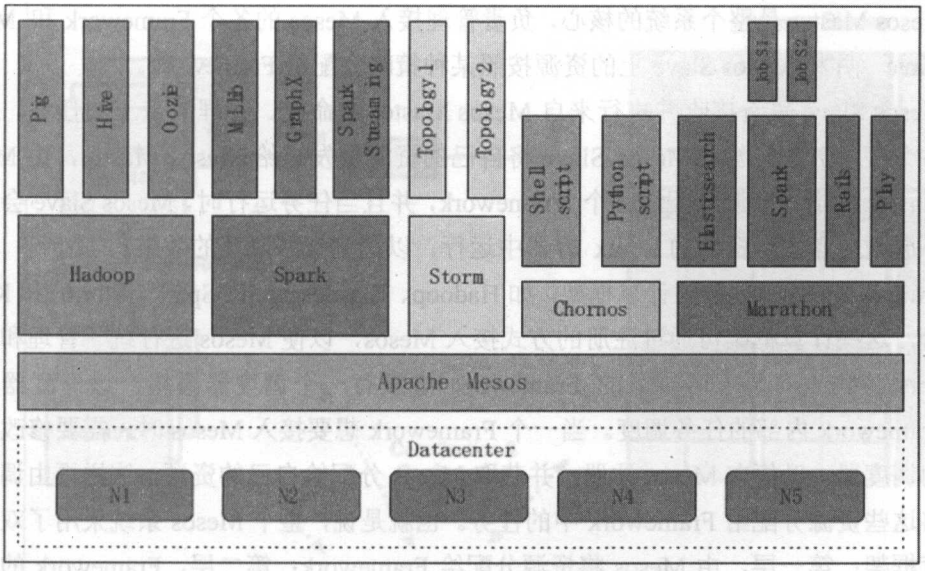


图 15-1 Mesos 的层次

15.2 Mesos 的架构

Mesos 的架构如图 15-2 所示，主要的角色有 Mesos Master、Mesos Slave、Framework 和 Executor。

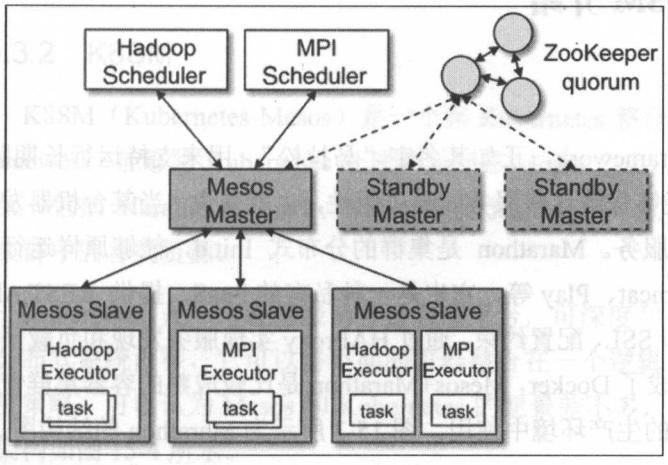


图 15-2 Mesos 架构

- Mesos Master 是整个系统的核心，负责管理接入 Mesos 的各个 Framework 和 Mesos Slave，并将 Mesos Slave 上的资源按照某种策略分配给 Framework。
- Mesos Slave 负责接收并执行来自 Mesos Master 的命令、管理节点上的任务，并为各个任务分配资源。Mesos Slave 将自己的资源量发送给 Mesos Master，由 Mesos Master 决定将资源分配给哪个 Framework，并且当任务运行时，Mesos Slave 会将任务放到包含固定资源的 Linux 容器中运行，以达到资源隔离的效果。
- Framework 是指外部的计算框架，如 Hadoop、Elasticsearch、Spark、Storm 和 Kafka 等，这些计算框架可通过注册的方式接入 Mesos，以便 Mesos 进行统一管理和资源分配。Mesos 要求可接入的 Framework 必须有一个调度器模块，该调度器负责 Framework 内部的任务调度。当一个 Framework 想要接入 Mesos 时，需要修改自己的调度器，以便向 Mesos 注册，并获取 Mesos 分配给自己的资源，这样再由调度器将这些资源分配给 Framework 中的任务。也就是说，整个 Mesos 系统采用了双层调度框架：第一层，由 Mesos 将资源分配给 Framework；第二层，Framework 的调度器将资源分配给自己内部的任务。
- Executor 主要用于启动 Framework 内部的任务。由于不同的 Framework 启动任务的接口或者方式不同，当一个新的 Framework 要接入 Mesos 时，需要编写一个 Executor，告诉 Mesos 如何启动该 Framework 中的任务。

15.3 Marathon 和 K8SM 介绍

15.3.1 Marathon

Marathon 是一个 Mesos Framework，正如其名字“马拉松”，用来支持运行长期服务，比如 Web 应用等。Marathon 能够保证这些服务的高可用性，也就是说，当某台机器发生故障时，能够在其他机器上启动服务。Marathon 是集群的分布式 Init.d，能够原样运行任何 Linux 二进制发布版本，如 Tomcat、Play 等，它也是一种私有的 PaaS，提供 REST API 服务，实现服务的发现，有授权和 SSL、配置约束，通过 HAProxy 实现服务发现和负载平衡。更重要的是，Marathon 已经集成了 Docker，Mesos+Marathon 是比较成熟的容器集群管理解决方案，已经在众多知名企业的生产环境中使用。图 15-3 所示为 Marathon 的结构图。

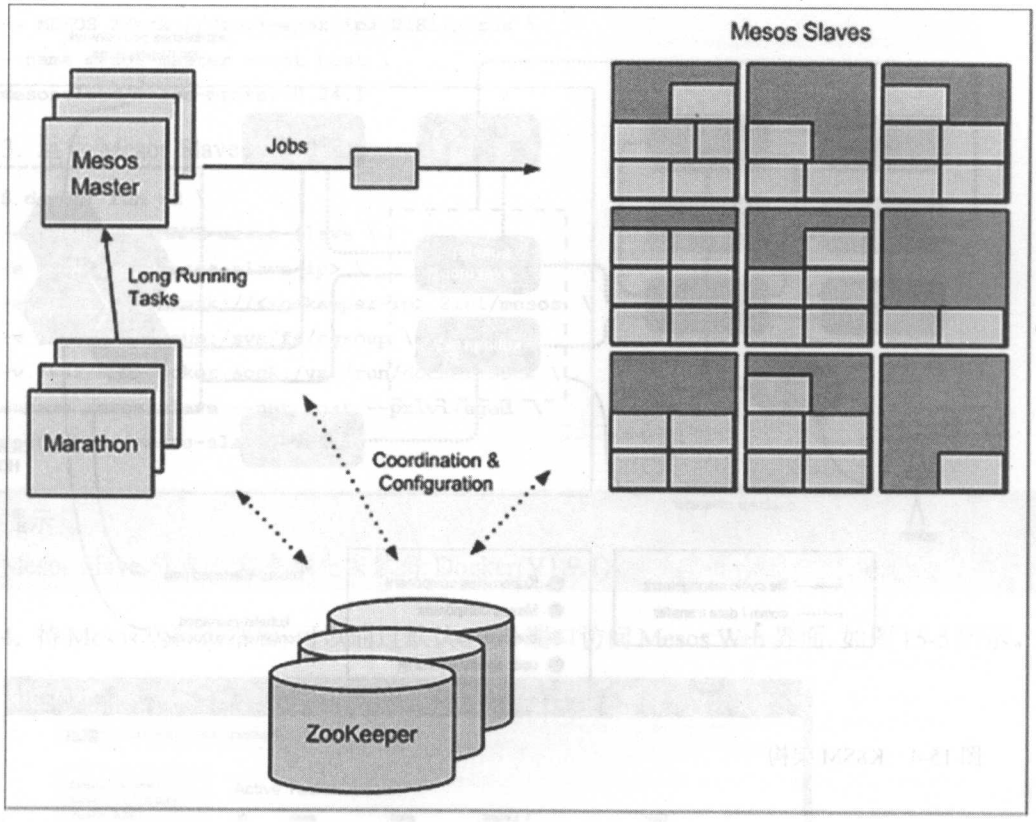


图 15-3 Marathon 结构

15.3.2 K8SM

K8SM（Kubernetes-Mesos）是一个将 Kubernetes 整合到 Mesos 中的项目，作为 Mesos Framework，能够将 Kubernetes 处理并运行在 Mesos 之上，且可以同任意数量的其他 Mesos 框架（包括 Marathon、Spark、Kafka 以及 Jenkins 等）实现同地协作，从而共享来自同一套集群中的各类资源。

Kubernetes 是一个轻量级的容器管理平台，可深度和灵活地进行容量编排，而 Mesos 是分布式系统内核，它可以将不同的机器整合在一个逻辑计算机中，有着非常优秀的资源调度策略，可以认为 Mesos 和 Kubernetes 的愿景差不多，而 K8SM 则整合了两者，K8SM 的架构如图 15-4 所示。

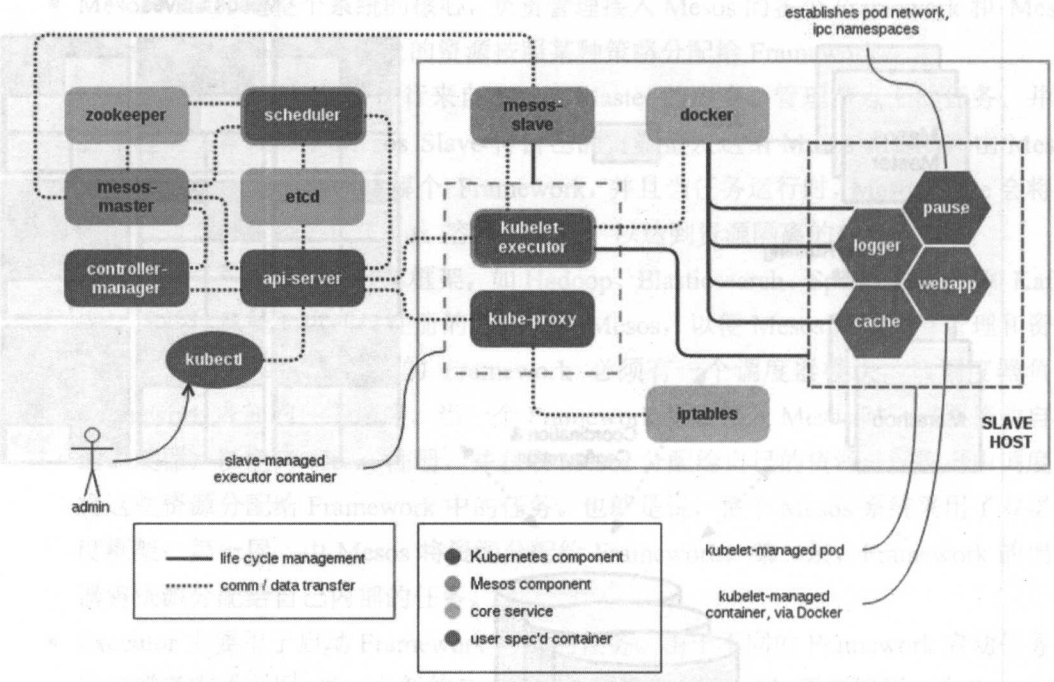


图 15-4　K8SM 架构

15.4　Mesos 实践

15.4.1　运行 Mesos

1. 运行 Zookeeper。

```
$ docker run -d \
--name zookeeper --net host \
mesoscloud/zookeeper:3.4.6
```

2. 运行 Mesos Master。

```
$ docker run -d \
-e MESOS_HOSTNAME=mesos-master \
-e MESOS_IP=<mesos-master-ip> \
-e MESOS_QUORUM=1 \
```

```
-e MESOS_ZK=zk://<zookeeper-ip>:2181/mesos \
--name mesos-master --net host \
mesoscloud/mesos-master:0.24.1
```

3. 运行 Mesos Slave。

```
$ docker run -d \
-e MESOS_HOSTNAME=mesos-slave \
-e MESOS_IP=<mesos-slave-ip> \
-e MESOS_MASTER=zk://<zookeeper-ip>:2181/mesos  \
-v /sys/fs/cgroup:/sys/fs/cgroup \
-v /var/run/docker.sock:/var/run/docker.sock \
--name mesos-slave --net host --privileged  \
mesoscloud/mesos-slave:0.24.1
```

提示

Mesos Slave 节点上需要预先安装好 Docker(V1.9.1)。

4. 待 Mesos 运行正常后，可以通过默认 5050 端口访问 Mesos Web 界面，如图 15-5 所示。

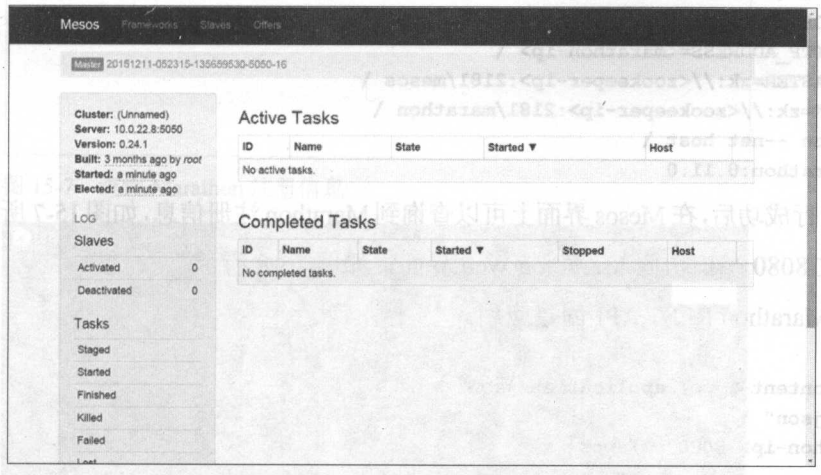


图 15-5　Mesos Web 界面

同时可以查询到 Mesos Slave，如图 15-6 所示。

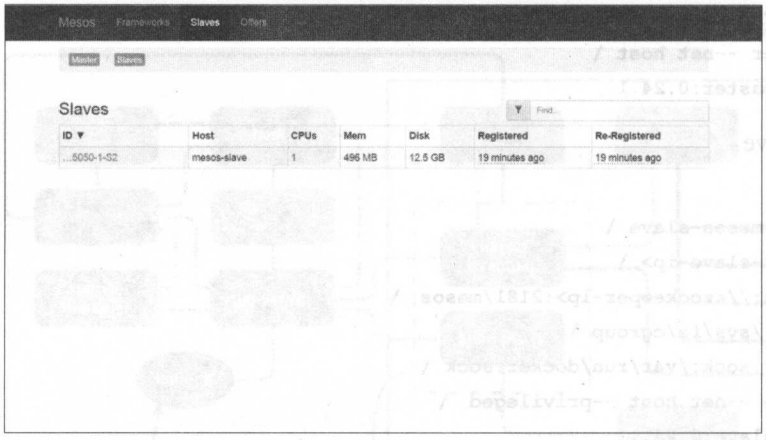


图 15-6　查询 Mesos Slave

15.4.2　运行 Marathon

1. 运行 Marathon。

```
$ docker run -d \
-e MARATHON_HOSTNAME=marathon \
-e MARATHON_HTTP_ADDRESS=<marathon-ip> \
-e MARATHON_MASTER=zk://<zookeeper-ip>:2181/mesos \
-e MARATHON_ZK=zk://<zookeeper-ip>:2181/marathon \
--name marathon --net host \
mesoscloud/marathon:0.11.0
```

2. Marathon 运行成功后，在 Mesos 界面上可以查询到 Marathon 注册信息，如图 15-7 所示。同时可以通过 8080 端口访问 Marathon Web 界面，如图 15-8 所示。

3. 通过调用 Marathon REST API 创建应用。

```
$ curl -v \
-X POST -H "Content-Type: application/json" \
--data "@app.json" \
http://<marathon-ip>:8080/v2/apps
```

其中 app.json 是应用的定义文件，内容如下：

```
{
  "id": "web-app",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 32.0,
```

```
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "python:3",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 8080, "hostPort": 0 }
      ]
    }
  }
}
```

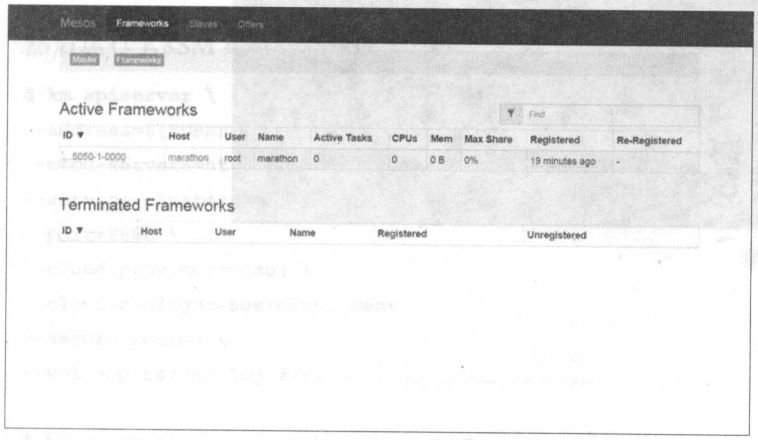


图 15-7　查询 Marathon 注册信息

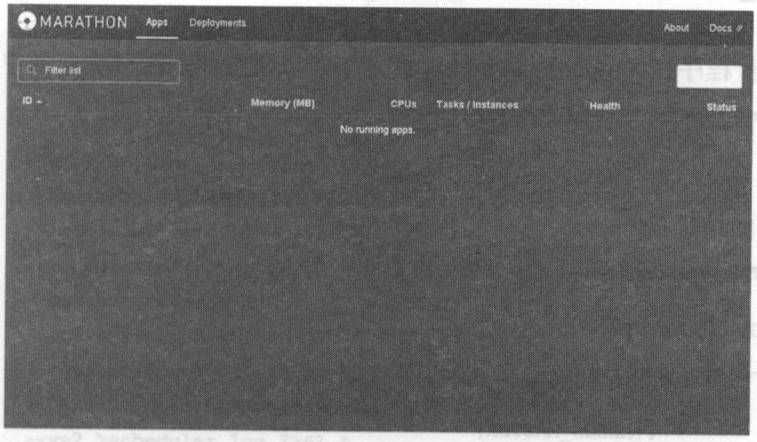


图 15-8　Marathon Web 界面

在应用定义文件中首先指定了应用的 ID、资源规格和运行实例数，在 container 属性中配置了 Docker 容器的运行参数。

应用创建成功后，Mesos Slave 将会运行起配置的 Docker 容器。另外，在 Marathon Web 界面可以查询到应用详情，如图 15-9 所示。

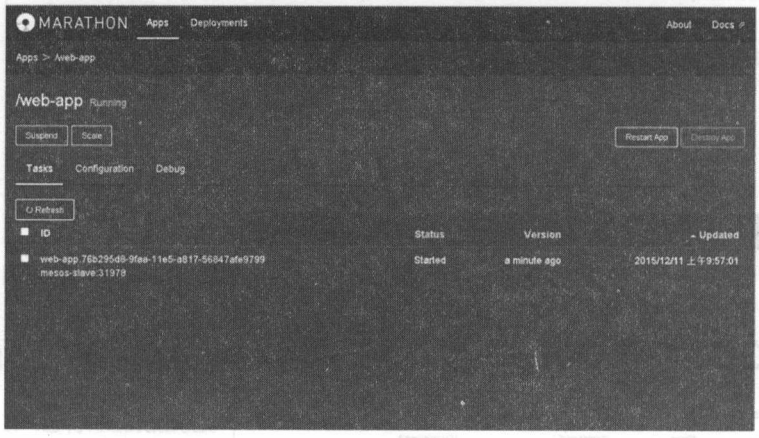


图 15-9　查询应用详情

15.4.3　运行 K8SM

1. 构建 K8SM。

```
$ git clone https://github.com/kubernetes/kubernetes
$ export KUBERNETES_CONTRIB=mesos
$ make
```

编译成功后，可执行程序在_output/local/go/bin 目录下可以设置到系统环境变量中：

```
$ export PATH="$(pwd)/_output/local/go/bin:$PATH"
```

2. 启动 K8SM。

设置 Mesos Master 的地址：

```
$ export MESOS_MASTER=<mesos-master-ip>:5050
```

然后配置文件 mesos-cloud.conf：

```
$ cat <<EOF >mesos-cloud.conf
[mesos-cloud]
      mesos-master        = ${MESOS_MASTER}
EOF
```

设置环境变量：

```
$ export KUBERNETES_MASTER_IP=$(hostname -i)
$ export KUBERNETES_MASTER=http://${KUBERNETES_MASTER_IP}:8888
```

首先运行 Etcd：

```
$ docker run -d \
-p 4001:4001 -p 7001:7001 \
--hostname $(uname -n) --name etcd \
quay.io/coreos/etcd:v2.0.12 \
--listen-client-urls http://0.0.0.0:4001 \
--advertise-client-urls http://0.0.0.0:4001
```

然后运行 K8SM 组件：

```
$ km apiserver \
--address=${KUBERNETES_MASTER_IP} \
--etcd-servers=http://127.0.0.1:4001 \
--service-cluster-ip-range=10.10.10.0/24 \
--port=8888 \
--cloud-provider=mesos \
--cloud-config=mesos-cloud.conf \
--secure-port=0 \
--v=1 >apiserver.log 2>&1 &

$ km controller-manager \
--master=http://${KUBERNETES_MASTER_IP}:8888 \
--cloud-provider=mesos \
--cloud-config=./mesos-cloud.conf  \
--v=1 >controller.log 2>&1 &

$ km scheduler \
--address=${KUBERNETES_MASTER_IP} \
--mesos-master=${MESOS_MASTER} \
--etcd-servers=http://127.0.0.1:4001 \
--mesos-user=root \
--api-servers=${KUBERNETES_MASTER_IP}:8888 \
--cluster-dns=10.10.10.10 \
--cluster-domain=cluster.local \
--v=2 >scheduler.log 2>&1 &
```

3. K8SM 运行成功后可以进行查询。

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get services
NAME              CLUSTER_IP      EXTERNAL_IP   PORT(S)     SELECTOR   AGE
k8sm-scheduler    10.10.10.132    <none>        10251/TCP   <none>     9s
kubernetes        10.10.10.1      <none>        443/TCP     <none>     40s
```

同时在 Mesos Web 界面上可以查询到 K8SM 注册信息，如图 15-10 所示。

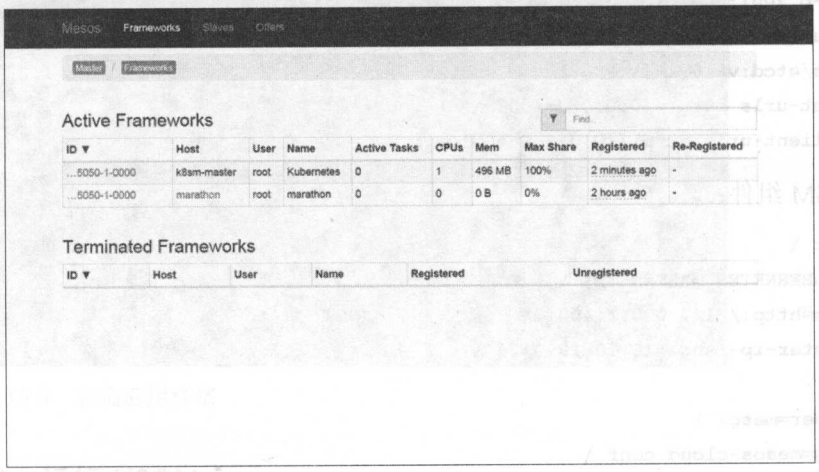


图 15-10　查询 K8SM 注册信息

4. 使用 K8SM 运行 Pod。

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 run nginx --image=nginx
replicationcontroller "nginx" created
```

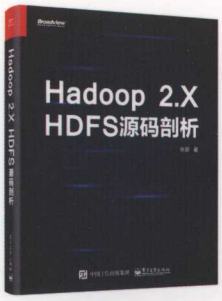
当有 Pod 创建后，K8SM Scheduler 向 Mesos 申请资源，然后 K8SM Scheduler 根据 Mesos 提供的资源绑定 Pod 到指定的 Mesos Slave。这时候，K8SM 自动会在 Mesos Slave 上运行 K8SM Executor，其中包括 Kubelet 和 Kube Proxy 组件。相应的，该 Mesos Slave 也会被作为 Kubernetes Node：

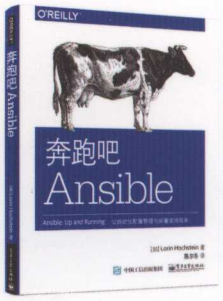
```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get node
NAME          LABELS                                   STATUS    AGE
mesos-slave   kubernetes.io/hostname=mesos-slave       Ready     22s
```

最后在 Mesos Slave 上，K8SM 将会运行起 Pod：

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get pod --selector run=nginx
NAME          READY     STATUS          RESTARTS   AGE
nginx-1dk7d   1/1       Runing          0          1m
```

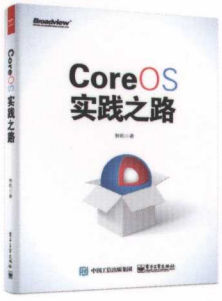
好书力荐

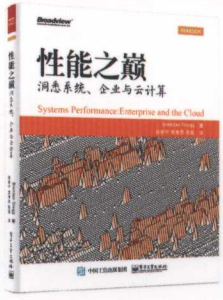

Hadoop 2.X
HDFS源码剖析


O'REILLY
奔跑吧
Ansible


微服务
架构与实践


Apache Mesos Essentials
Mesos
大数据资源调度与大规模容器运行最佳实践


CoreOS
实践之路


性能之巅
洞悉系统、企业与云计算
Systems Performance: Enterprise and the Cloud

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Kubernetes实战

容器技术的发展带来新变革，这也是DockOne.io社区创立的重要因素。在这个新技术的交流平台中，对于Kubernetes的讨论是非常活跃的。本书是一本理论与实践密切结合的书籍，作者有着丰富的实战经验，无论是初学者还是资深用户，都能够从书中获取知识与技能。相信这本书能在推动Kubernetes在国内的发展中起到重要作用！

• DockOne.io发起人 李颖杰

如果IaaS是云的第一次世界大战，那么PaaS即将迎来云的第二次世界大战。IaaS单纯改变基础设施资源的供给方式，PaaS将全面变革IT开发和运维的生产流程。本书详细介绍了第三代PaaS代表技术之一的Kubernetes，深入浅出，值得一读。

• 资深运维从业者，杭州云霁科技有限公司CEO 智锦

容器无疑是下一个技术热点，容器相关的图书也出了不少，但是这些书中侧重原理及代码的较多，侧重实践性的比较稀缺。《Kubernetes实战》这本书填补了这方面的空白。技术在生产环境落地才能带来收益，目前许多公司正在如火如荼地推行容器技术，《Kubernetes实战》这本书出现得非常及时，相信通过这本书精湛的理论讲述及丰富的案例，可以让大家少走许多弯路。

•《深度实践KVM》作者 肖力

Kubernetes作为优秀的开源容器集群管理系统从出现后便备受青睐，IT巨头的支持，架构师的实践，让Kubernetes不断完善并快速成长。本书结合实战案例系统地讲解了Kubernetes的方方面面，从基础架构到生态系统，有助于快速掌握Kubernetes并运用到实践中，推荐阅读。

• 网宿CDN运维总监 吴振永

作为Google系的开源软件，Kubernetes在开源之时就备受关注。InfoQ也在很早之前就开始向社区普及推广Kubernetes的架构和实践。本书就Kubernetes的架构、组件、网络、应用等内容做了详细介绍，先从为什么有Kubernetes开始，接着讲它是什么，以及如何使用，由浅入深，渐入佳境，推荐阅读。

• InfoQ主编 郭蕾

在当前云计算这股PaaS攻占IaaS的大潮流中，轻量级容器技术以及轻量级云平台，已然成为绝对的主角。其实单从技术上说，理解Docker容器不难，而要将网络、存储、调度、大规模多实例的应用管理及微服务框架统一在一起，则需要一个非常优秀的架构。Kubernetes作为Google多年大规模容器管理技术的开源版，非常值得大家学习借鉴。本书不仅可以作为Kubernetes入门的优秀指导，更难能可贵的是，它对于涉及的特定概念及操作细节都描述得非常细致，可以有效地解决实际使用中遇到的问题，非常适合作为Kubernetes的使用手册常备。在这里也希望该书可以引导读者更深一步地去开发构建最适合自己的云平台。

• 华为PaaS平台架构师 唐盛军

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：刘 舫
封面设计：侯士卿

上架建议：Docker/容器管理

ISBN 978-7-121-28372-7
9 787121 283727
定价：69.00元